A SYSTEMS PROGRAMMING LANGUAGE

(ASPL)

October 1, 1971

Tom Blease
Doug Jeung
Gerry Bausek

TABLE OF CONTENTS

I. INTRODUCTION

COMMENTS ON ASPL

ASPL is a high level procedure oriented programming language designed for use in implementation of all standard software for the ALPHA computer. ASPL contains facilities for performing arithmetic, logical, byte, and bit operations on a variety of data structures.

The ASPL design reflects the ALPHA architecture. Declaration facilities provide the programmer with the capability for use of all defined addressing modes. The assemble statement allows the user to emit all defined ALPHA machine codes.

The ASPL compiler is highly modular in structure. New constructs can be defined and added to the syntax with relative ease.

The reader should note that while ASPL is a high level language, many machine dependent functions are defined in the language. The user should become familiar with the ALPHA architecture and instruction set before analyzing the syntax.

ASPL Syntax Notation

A BNF (backus normal form) description of the ASPL language is given in the following text. It completely defines the syntax for the language and should be referred to if questions regarding syntax arise.

1) The syntax is described through the use of metalinguistic symbols. These symbols have the following meanings:

    < > Left and right broken brackets are used to contain one or more characters representing a metalinguistic variable. The value of each metalinguistic variable will be given by a metalinguistic formula.

    ::= The symbol ::= means "is defined as" and separates the metalinguistic variable on the left of the formula from its definition on the right.

    | The symbol | means "or". This symbol separates multiple definitions of a metalinguistic variable.

Example --

```
<Leftpart>  ::=  <A><B>|<C>|<Leftpart><D>
```

The above metalinguistic formula is read as follows:

Leftpart is defined as A followed by B, or C, or leftpart followed by D.

2) Productions of the following form are assumed, but not explicitly displayed, for the oft-recurring list construct:

```
<notion list>  ::= <notion>|<notion list>,<notion>
```

To conveniently express the type-attribute of a syntactic entity the following notation is employed.

The occurrence of the symbol T in a syntactic rule specifies that this symbol must be replaced consistently by any one of a set of English words. Below is a table that defines the replacing words for each of the symbols: T, $T_\emptyset$, T_1, T_2, T_3, T_4

T :	integer, real, logical, byte, double, long
$T_\emptyset$:	integer, logical, byte
T_1:	integer, logical
T_2:	integer, real, logical, double, long
T_3:	integer, real, byte, double, long
T_4:	integer, real, long

Two examples illustrate the replacement of T symbols:

1) The syntactic rule

<$T_\emptyset$ array id> ::= <identifier>

corresponds to

<integer array id> ::= <identifier>
<logical array id> ::= <identifier>
<byte array id> ::= <identifier>

2) The notation

<$T_\emptyset$ variable> ::= <$T_\emptyset$ simpvar id>|
<T_1 pointer id>

corresponds to

<integer variable> ::= <integer simpvar id>|
<integer pointer id>

<logical variable> ::= <logical simpvar id>|
<logical pointer id>

<byte variable> ::= <byte simpvar id>

II. PROGRAM COMPONENTS

PROGRAM COMPONENTS

Syntax --

```
<program>               ::=  <global head> <main body> .
<global head>           ::=  BEGIN <data group> <procedure group>
<main body>             ::=  <compound tail>
<compound tail>         ::=  <statement> END |
                             <statement> ; <compound tail>
<compound statement>    ::=  BEGIN <compound tail>
<subprogram>            ::=  BEGIN <subdata group><procedure group>END
```

Semantics --

An ASPL program is an ordered collection of data declarations and procedure declarations followed by a main body.

An ASPL subprogram is a collection of procedure declarations. A subprogram is defined to be a program with an empty data group and empty main body.

The compiler will produce an object file containing the object code, globally allocated storage, and various bits of information for the loader.

Example --

```
1.)  BEGIN
       INTEGER I,J;
       PROCEDURE P(A); VALUE A; INTEGER A;
           I ← 2*A;
       P(2*J);
     END.


2.)  BEGIN
       PROCEDURE P1;---;
       PROCEDURE P2;---;
       :
       :
       PROCEDURE PN;---;
     END.

3.)  BEGIN
     END  .- null program
```

COMPILER CONTROL OPTIONS

Syntax --

```
<compiler control command>   ::= $ <control command part>
<control command part>       ::= <control command>|
                                 <control command part>, <control command>
<control command>            ::= ADR|CLIST|INNERLIST|
                                 LIST|UNLIST|
                                 COMPILE|NOCOMPILE|PAGE|
                                 SEGMENT = &lt;segment name&gt;|
                                 MAIN = <program name>|
                                 VOID = <exclusive of field>|
                                 INPUT = <source file name >|
                                 CODE = <object file name>|
                                 EDIT|NEW|OTT|SUBPROGRAM
```

Examples --

```
$ FILE=TAB1 CARD ADR CLIST
$ FILE=SCR1 TAPE EDIT NEW
$ UNLIST LIST
```

Semantics --

The compiler control options allow the user to specify modes of input and output, and what form of listing will be emitted. A compiler control option may be used anywhere in the source program. Upon encountering a $, the remaining characters in the source image are assumed to contain control commands. Upon completing the image, the next sequential source image will be read, and the compiler will resume normal compilation. Control commands are delimited by one or more blanks.

MNEMONIC	FUNCTION
ADR	Print a line after each declaration, showing the addressing mode and displacement of the declared variable.

MNEMONIC	FUNCTION
CLIST	Print code emitted after each procedure and outer block. Prints code address and 8 instructions per line.
INNERLIST	Print innerlist of code emitted by compiler.
LIST	Print a line on the printer containing the source image, tape sequence number if appropriate, lexicographical level and current code address. This option is used by default.
UNLIST	Turn off ADR, CLIST, LIST and INNERLIST options.
CARD	Source images will be read from card reader.
OTT	All printing will be directed to the input terminal.
SCOPE	Source images will be read from the teletype. This option is used by default.
TAPE	Source images will be read from magnetic tape.
EDIT	Edit tape images with card images during compilation. Columns 73-8Ø of the current tape image are compared with columns 73-8Ø of the current card image. Given TF = tape compare field, CF = card compare field then 1. CF < TF use card image (insert) {Read next card image. 2. CF = TF use card image and {skip tape image (replacement) Read next card image. 3. CF > TF use tape image. The comparison is determined by performing an 8 byte ASCII compare on CF/TF.

MNEMONIC	FUNCTION
NEW	Generates a new source tape.
DISC=XXXX	Source images will be read from the disk file named XXXX. If XXXX = TAPE, the images will be assumed to be read from magnetic tape.
FILE=YYYY	Object code will be written on a disk file named YYYY.
VOID <exclusive of fields>	This option is valid only when the EDIT option is being used. Columns 73-80 of this card are treated as described above. The notation <exclusive of field> defines an 8 byte ASCII field, separated from the symbol VOID by at least one blank. Tape images will be skipped while <exclusive of field> > TF.

Example --

```
$ VOID 01250000                                         00101135
```

INPUT = <source file name>	Source images will be read from the file specified.
CODE = <object file name>	Object code will be written on file specified.
COMPILE	If a NOCOMPILE was encountered previously then compilation will begin again; otherwise ignored.
NOCOMPILE	Stops compiling and treats source images as comments until a COMPILE option is encountered.
PAGE	Ejects a page on line printer
SEGMENT = <segment name>	Every segment must have a name. Each occurrence of the SEGMENT option starts a new segment. The default name is SEG'.
MAIN = <program name>	Every main program must have a name. The default name is OB'.
SUBPROGRAM	Either subprogram is specified or it is assumed to be program mode. Subprogram means no global data storage will be assigned by the compiler and no outer block code is permitted.

COMMENT CONSTRUCTS

Syntax --

<comment> ::= COMMENT <any sequence of ASCII characters excluding ;> ; |
<< {any sequence of ASCII characters} >>

Semantics --

Two forms of comment insertion are defined in ASPL.

Form one, COMMENT, will be also used as a control statement for the ALPHA flow charter and text editior packages. Form 1 is treated as a null statement, and can be used anywhere a statement is syntactically valid.

Form 2 is allowed anywhere in the program.

Examples --

```
COMMENT CONTROL:  MESSAGE ;
<<THIS IS ALSO A COMMENT>>
```

Note: The following statements are equivalent.

```
IF X<Y THEN COMMENT; GO TO L;
IF X<Y THEN; GO TO L;
```

GENERAL

Delimiters

A blank is recognized as a delimiter in ASPL. The only exception is the occurrence of a blank in a string constant.

III. NUMBERS, STRINGS, IDENTIFIERS

CONSTANTS

Syntax --

```
<constant>      ::=  <number> | <string>

<number>        ::=  <integer>|
                     <real number>|
                     <double integer>|
                     <long real number>|
                     <logical value>
```

Semantics --

Strings are considered as type byte. No extended precision hardware operators are defined for handling long real numbers. The inclusion of long real numbers is a generalization useful for the development of library routines requiring extended precision. Calculations requiring extended real precision will be handled by calling library routines. Compatibility between constants and identifiers will be handled by the compiler with respect to type and precision.

INTEGER CONSTANTS

Syntax --

```
<integer>              ::=  <unsigned integer>|
                            <sign><unsigned integer>

<decimal integer>      ::=  <digit>|
                            <decimal integer><digit>

<unsigned integer>     ::=  <decimal integer>|<based integer>|<composite integer>

<digit>                ::=  0|1|2|3|4|5|6|7|8|9

<sign>                 ::=  +|-

<based integer>        ::=  %<base part><base digit>|
                            <based integer><base digit>

<base part>            ::=  (<base>)|<empty>

<base>                 ::=  any number of the set {2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}

<empty>                ::=  a character string of zero length

<base digit>           ::=  any of the digits from 0 to <base>-1  inclusive, taken
                            from the set {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}

<composite integer>    ::=  [<integer field list>]

<integer field>        ::=  <number of bits>/<decimal integer>|
                            <number of bits>/<based integer>|
                            <number of bits>/<composite integer>

<number of bits>       ::=  <decimal integer>

<double integer>       ::=  <integer> D
```

Semantics--

An empty base part of a based integer is taken to be 8 (octal) by default. Composite numbers are right-justified binary integers formed through left to right concatenation of the binary integers defined by the integer fields. Any unspecified leading bits will be filled with zeros. Binary integers exceeding the field size will be truncated on the left. For example, the integer field 6/%177 will be truncated to %77. Any unusual condition will be flagged by the compiler.

Care in the use of blank as a delimiter should be exercised when specifying based integers.

The based integer

    %(16)ABCD

is not equivalent in either type or value to the based integer

    %(16)ABC D.

The blank inserted in the second example specifies that the constant is a double integer.

Examples--

<decimal integer>	123941
<based integer>	%1777, %(2)Ø11Ø1Ø111, %(11)ØA19
<composite integer>	[3/2,12/%5252], [2/211, 15/[3/%(2)1Ø1, 12/Ø], 1Ø/123] D
<integer>	-123, -%(2)111ØØ1Ø

REAL CONSTANTS

Syntax--

<real number>	::= <unsigned real number>| <sign><unsigned real number>
<unsigned real number>	::= <fraction>| <decimal integer> E <power>| <fraction> E <power>| <composite integer> E| <based integer> E
<fraction>	::= <decimal integer> .| .<digit>| <fraction> <digit>
<power>	::= <decimal integer>| <sign> <decimal integer>
<long real number>	::= <unsigned long real number>| <sign> <unsigned long real number>
<unsigned long real number>	::= <decimal integer> L <power>| <fraction> L <power>| <composite integer> L| <based integer> L

Examples--

1.3214 , .1Ø21 , -1.1Ø5E-21
[3/7,6/32,25/%(16)ABCDEB] L
% (6)1235415E , + 1.3971
9321.Ø15E28

LOGICAL CONSTANTS

Syntax --

<logical value> ::= TRUE|FALSE|<integer>

Semantics --

Logical quantities are sixteen bit positive integers. Operations on logical values are defined in the hardware for addition, subtraction, multiplication, and division. A logical value is considered true if its value is odd, false if its value is even, i.e. only the last bit is checked. However, assignment of TRUE or FALSE corresponds to assigning the numeric value -1 or 0, respectively.

STRING CONSTANTS

Syntax --

<string> ::= "<character string>"

<character string> ::= <character> |
<character string><character>

<character> ::= {a member of the set of ASCII character representations}

Semantics --

If a quote (") is to appear within a character string, it is represented by a pair of quotes.

Examples --

"THE CHARACTER """ IS A QUOTE MARK"
"A NORMAL STRING WOULD LOOK LIKE THIS"

IDENTIFIERS

Syntax --

```
<identifier>            ::= <letter>|
                            <identifier><letter>|
                            <identifier><digit>|
                            <identifier>'

<T simpvar id>          ::= <T identifier>

<T identifier>          ::= <identifier>

<T pointer id>          ::= <identifier>

<T array id>            ::= <identifier>

<procedure id>          ::= <T proc id>|
                            <proc id>

<T proc id>             ::= <identifier>

<proc id>               ::= <identifier>

<entry id>              ::= <identifier>

<label id>              ::= <identifier>

<switch id>             ::= <identifier>

<equate id>             ::= <identifier>

<define id>             ::= <identifier>

<subroutine id>         ::= <T subr id>|
                            <subr id>

<T subr id>             ::= <identifier>

<subr id>               ::= <identifier>
```

Semantics --

The symbol T must be replaced with:

integer
logical
byte
real
long
double

The form of an identifier must be a sequence of less than 16 contiguous alphanumeric characters. The first character must be a letter. Identifiers longer than 15 characters will be truncated on the right. The attributes of an identifier are determined by a declaration, not in any way by the form of the identifier. Apostrophes may also be used in an identifier, but may not be the first character.

The compiler will assume 7 bit ASCII representation. Lower case letters are allowed but will be converted to upper case letters except when they appear in strings.

Section VIII.2 contains a list of reserved symbols. These symbols may not be used as identifiers since they have an implied meaning as defined in the syntax.

Examples --

MATRIX
MAT1
A123B
X
XYZ
AN'IDENTIFIER

IV. DECLARATIONS

SCOPE OF DECLARATION AND ADDRESSABILITY

Variables assigned DB-relative locations or the X-register are addressable within the scope of the entire program (that is, their values are accessible, although their names may be undefined in various segments of the program).

Variables assigned Q relative locations are addressable within the scope of the procedure in which they are declared.

Variables assigned S relative locations are addressable within the scope of the current S register setting.

ADDRESS ASSIGNMENTS

Address assignments specify the addressing convention and storage allocated for variables. Variables declared globally, are assigned the DB relative addressing convention. Variables declared in the scope of a procedure , local variables, are assigned the Q+ addressing convention. The number of locations allocated depends on the data structure and type being declared. The following diagram will be useful in explaining the implications of address assignments.

Assume that the declarations below have been made. When procedure PROC is called and executed, the data area will look like the figure on p.IV.2.2.

```
BEGIN
  INTEGER ARRAY A(Ø:72);
  INTEGER I;
  LOGICAL ARRAY AA(Ø:9) = DB;
PROCEDURE PROC;
  BEGIN
    INTEGER X,Y;
    BYTE ARRAY C(Ø:11);
    OWN LOGICAL ARRAY B(3:7);
```

DATA AREA

DB		
+Ø	12 (data label for ARRAY A)	PRIMARY DB DATA AREA:
1	cell allocated for INTEGER I	DB [Ø:N] directly addressable relative
2		to DB register, where Ø<=N<=255
	direct ARRAY AA	
N=11		
12		SECONDARY DB DATA AREA:
		DB [N+1:Q_I] accessible through
		data label, (indirect addressing),
	storage for ARRAY A	where Q_I is the initial Q-register
		setting
84		
85		
	storage for OWN ARRAY B	
89		
Q_I	- - - -	INITIAL Q-REGISTER SETTING
	- - - -	
	- - - -	
Q		Q-REGISTER SETTING FOR CURRENT PROCEDURE
+1	INTEGER X	(Procedure PROC in this example)
2	Y	
3	Q-DB+5 (data label for ARRAY C)	
4	85 (data label for OWN ARRAY B)	Local Variable Storage
5		
	storage for ARRAY C	
S		S-REGISTER SETTING
		Remaining User Stack
Z		

Simple Variable and Pointers

A simple variable or pointer is allocated the next sequential location(s) relative to DB or Q as specified above. The number of locations allocated is:

a) One for integer, logical, byte or pointer variables.

b) Two for double or real variables.

c) Three for long variables.

Example: The global declaration INTEGER I would allocate the location DB+N to the variable I. The next assignable DB relative location would be DB+(N+1).

Arrays

I. Indirectly addressed through a data label:

Normally, an array is allocated storage in either the secondary DB data area or in the stack following local variable allocation. The data label through which the array is accessed is assigned a single location relative to DB if global, or Q if local.

If the array bounds are undefined and the reference part is empty, the user will be required to allocate his own storage and initialize the location allocated for the data label with the address of the allocated space.

Examples of code for storage allocation will be found in section IV.7.

If the array bounds are dynamic, the upper and lower bound will be computed, and storage will be allocated in the stack upon procedure entry.

The quantity of space allocated for an array is dependent on its type.

a) N words for integer or logical arrays of N elements.
b) 2*N words for double or real arrays of N elements.
c) 3*N words for long arrays of N elements.
d) (N+1) DIV 2 for byte arrays of N elements.

II. Directly addressable:

ASPL allows the user to specify that an array will be directly addressable by using the notation =DB or =Q as defined in section IV.7.1. Care should be exercised, however, in the use of direct arrays, since the available directly addressable range is rather limited. Directly addressable arrays are allocated the next M sequential locations relative to DB or Q, where M is the number of words to be allocated.

ADDRESS REFERENCES

Address references allow the programmer to equate variables to other variables or equate an address relative to an addressing convention. Generally, no storage is allocated when an address reference is specified. Exceptions will be noted. There are three forms of address referencing:

Form 1.

<identifier reference> | <identifier reference> <sign> <unsigned integer>

The declared simple variable or pointer is assigned the addressing convention and address of the referenced identifier, adjusted by the value of the sign-and-integer if present.

Examples:

Assume the variable A has been assigned the location DB+5. Then we would have the following:

REAL	B = A would assign B the location DB+5.
INTEGER	BR=A+1 would assign BR the location DB+6.
POINTER	P=A-5 would assign P the location DB+Ø.

Form 2.

DB+N | Q+N | Q-N | S-N | X where N is an unsigned integer.

The variable is assigned the address N relative to the specified addressing convention.

Note that Q relative addressing may be negative and S relative addressing is always negative or zero. The X address reference is valid only for simple variables of type integer or logical.

Variables equated with the X-register are assumed to all have the same value. Since the X-register is used in all array indexing plus numerous other calculations, the current value contained in X is the value assigned all variables equated to X. The compiler will not save the value of X. It will be the user's responsibility to maintain integrity for variables equated to the X-register if he so desires.

Arrays specifying this option will be accessed directly if the undefined bounds specifier is an *, with the array base being taken as the displacement N from the specifier register -- excluding X, of course. If the undefined bound specifier is an @, the array will be accessed indirectly through the assigned location.

Examples:

```
INTEGER    POINTER    SØ = S-Ø;
REAL    R = DB+25;
INTEGER    X=X;
```

Form 3.

<indexed ident ref> ::= <array id>|<array id>(<integer>)|<pointer id>|<pointer id>(<integer>)

This form is restricted for use with arrays only. It is used to equate an array with a previously declared array or pointer.

This form may also require the allocation of a cell for a data label, through which the declared array will be accessed. This will occur if the index part is non-empty, or if the referenced variable specifies indirection and its data label is incompatible with the form of data label required for accessing the declared array. A byte data label has a different format than a word data label (i.e. byte address vs. word address).

Examples:

```
1.) INTEGER ARRAY IA(Ø:5);
2.) LOGICAL ARRAY LA(*) = IA;
3.) POINTER P;
4.) BYTE ARRAY BA(*) = IA;
5.) BYTE ARRAY BAX(*) = BA(3);
6.) ARRAY LAX(*) = P;
```

Comment on examples on previous page:

Example 2.) LA takes on the same convention, address, and indirection as IA.

Example 4.) BA requires the allocation of a location since their data labels are incompatible.

Example 5.) BAX requires the allocation of a location since an index is specified.

Example 6.) LAX takes on the same convention, address, and will be accessible through the same data label assigned to P.

STATIC INITIALIZATION

Static initialization of simple variables is specified by following the identifier with a ← and a constant of appropriate type. This allows a variable to be compiled with an initial value of the programmer's choosing.

Static initialization of arrays is only allowed for global, local <u>own</u> or local PB-relative arrays with defined dimensions. The elements of the initialization list are separated by a comma. Note that local arrays may only be statically initialized if declared PB-relative or own. Global PB relative arrays are not allowed, as shown on p.IV.7.1. Byte variables may be initialized with string, integer, or logical values. If the initialization value requires more than eight bits for its representation, a warning message will be emitted.

Strings may also be used to initialize variables of non-byte types.

Examples:

```
INTEGER L ← - 144, J ← Ø, K ← %255;
BYTE B ← "$"
REAL X ← 3.1459, Y ← 34156;
LOGICAL LX ← %177777;
BYTE ARRAY EMESS (Ø:131) ← "**** ERROR NO. XXX *****";
INTEGER ARRAY LINKTAB (Ø:35) ← Ø,231, -4,126, ----;
OWN ARRAY OTAB (5:1Ø) ← Ø,1,2,3,4,5;
```

Note: Only the last array specified in an array declaration list can be initialized.

GLOBAL-EXTERNAL OPTION

Local variables assigned the attribute 'EXTERNAL' will be linked to the appropriate location by the loader. An 'EXTERNAL' variable must match in name and type a global variable which has been assigned the attribute 'GLOBAL'.

'EXTERNAL' variables may not be used as the object of an address reference, nor may they be given an address reference or assignment, nor may they be statically initialized.

The actual address at run time will be the global address assigned the 'GLOBAL' variable; the loader will flag a load error if no match is found.

Of course, the attribute 'GLOBAL' may only be used in a global declaration.

TYPE ATTRIBUTE

The type assigned a variable defines the allowable set of ALPHA instructions which may operate on the variable. A thorough definition of type will be found in the ALPHA external reference specification. A simplified coverage is given here.

Type. Six declarators are defined for type assignment; the values assigned variables with the following types are:

a.	INTEGER	(single precision positive and negative integral values, including zero. Sixteen bit, two's complement representation).
b.	REAL	(single precision positive and negative floating point values, including zero. Thirty-two bit sign + magnitude representation).
c.	DOUBLE	(double precision positive and negative integral values, including zero. Thirty-two bit, two's complement representation).
d.	LONG	(extended precision positive and negative floating point values, including zero. Forty-eight bit sign + magnitude representation).
e.	LOGICAL	(sixteen bit positive integral representation).
f.	BYTE	(eight bit integral representation).

DECLARATIONS

Syntax --

```
<data group>              ::=  <data group> <data declaration> ; |
                               <empty>

<data declaration>        ::=  <simpvar declaration> |
                               <pointer declaration> |
                               <array declaration> |
                               <label declaration> |
                               <switch declaration> |
                               <entry declaration> |
                               <equate declaration>|
                               <define declaration>

<procedure group>         ::=  <procedure group> <subroutine declaration>|
                               <proc group>

<proc group>              ::=  <proc group> <procedure declaration>|
                               <intrinsic group>

<intrinsic group>         ::=  <intrinsic group> <intrinsic declaration>|
                               <empty>

<subdata group>           ::=  <subdata group> <subdata declaration>|
                               <subdata declaration>| <empty>

<subdata declaration>     ::=  <simpvar declaration>|
                               <pointer declaration>|
                               <array declaration>|
                               <equate declaration>|
                               <define declaration>
```

Semantics --

Declarations are provided in the language for giving the compiler information about the constituents of the program, such as array sizes, the types and values variables may assume, labels, procedures, etc. Each such constituent must be named by an identifier, and all identifiers must be declared before they are used. Labels may be declared implicitly by their appearance in a switch declaration, a GOTO statement, or of course, by labeling a statement.

Elements of a subdata group which imply storage locations must either be external or have address references.

ARRAY DECLARATIONS

Syntax--

```
<array declaration>          ::= <global array dec>|
                                 <local array dec>

<global array dec>           ::= <atype> ARRAY <g-array dec list>|
                                 GLOBAL <atype> ARRAY <G-dec list>

<local array dec>            ::= <atype> ARRAY <L-array dec list>|
                                 EXTERNAL <atype> ARRAY <E-dec list>|
                                 OWN <atype> ARRAY <own array dec list>

<atype>                      ::= <type>|<empty>

<G-dec>                      ::= <identifier>(<db>)=DB<array init list>|
                                 <identifier>(<db>)<array init>

<g-array dec>                ::= <G-dec>|
                                 <identifier>(@)<indirect base register reference>|
                                 <identifier>(<udb>)<reference part>

<E-dec>                      ::= <identifier>(<udb>)

<L-array dec>                ::= <identifier>(<db>)=Q|
                                 <identifier>(<db>)|
                                 <identifier>(<db>)=PB←<listelmt>|
                                 <identifier>(<vb>)|
                                 <identifier>(<udb>)<reference part>|
                                 <identifier>(@)<indirect base register reference>

<own array dec>              ::= <identifier>(<db>)<array init>

<db>                         ::= <integer>:<integer>     {defined bounds}
<udb>                        ::= *                       {undefined bounds}

<vb>                         ::= <T1 simpvar id>:<T1 simpvar id>{variable bounds}

<reference part>             ::= <var reference>|
                                 =<indexed ident reference>

<indexed ident reference>    ::= <array id>|<array id>(<integer>)
                                 <pointer id>|<pointer id>(<integer>)
```

```
<array init>                   ::= <empty>|
                                  <listelmt>
<listelmt>                     ::= <initial value>|
                                  <decimal integer>(<listelmt>)| <listelmt list>
<var reference>                ::= <empty>|
                                   = <identifier>
                                   = <identifier><sign><USI>
                                   = <base register reference>
<base register reference>      ::= DB + <USI>
                                   Q + <USI>
                                   Q - <USI>
                                   S - <USI>
<USI>                          ::=<unsigned integer>
<indirect base register reference> ::= <empty>|
                                        =<base register reference>
```

Semantics--

An array declaration specifies one or more identifiers to represent arrays of subscripted variables. An array declaration may define the following information about an array identifier.

1. The bound specification which determines the range of indexing.
2. The type of values the array elements will take on.
3. The storage allocation method.
4. Array initialization if specified.
5. The access mode, direct or indirect.

If the array has been specified OWN, storage will be allocated and initialized by the loader. The loader will also guarantee that the Q+ relative cell allocated for the array data label will be initialized upon procedure entry. OWN arrays are accessible only by the procedure in which they are declared or by procedures called by the declarer to which the declarer has passed the array by reference.

Note that global PB-relative arrays are not allowed and that local arrays may only be statically initialized if declared PB-relative or own.

Bound checking code is not emitted by the compiler.

An empty type attribute will assign the array type LOGICAL by default.

If indirect addressing is specified, the data label will point at the zero element of the array. If the lower bound is greater than zero, or the upper bound is less than zero, then the zero element does not reside in the allocated space.

If direct addressing is requested, the displacement relative to the specified register will correspond to the zero element of the array.

An upper bound must not be smaller than the corresponding lower bound.

Dynamic bounds are evaluated once upon entry into the procedure. These bounds can depend only on values which are non-local to the procedure for which the ARRAY declaration is valid.

If the array specifies a PB relative reference, the initialization list will be copied into the code string preceding all executable code for the procedure. The range of the array will be the length of the list. Arrays so specified can not be used as the object of a store operation.

Arrays specifying undefined bounds will be handled as follows:

1. If the array has the attribute EXTERNAL, it will be linked as described in the section on GLOBAL - EXTERNAL.
2. If the array specifies an empty reference part, the array will be indirectly accessed through a DB+ or Q+ relative location assigned by the compiler. The user must allocate his own storage and initialize the assigned location with a corresponding data label.
3. The remaining uses of an undefined bound are described in detail in the section on address references.

The symbol T_1 must be replaced as given on p.I.2.2.

Examples--

```
LOGICAL         ARRAY     T(*) = S-1, A(Ø:4) = DB ← Ø,1,2,3;
BYTE   ARRAY        B(*) = A,    BITE (1:6)←"******";
REAL   ARRAY         RA(Ø:5) ← 1.123,Ø,-3.14159, 1,-.ØØØ3;
INTEGER ARRAY           IA(LB:UB), IA1(Ø:UB), IA2(*) = B
ARRAY  LL(*),          LB (Ø:1ØØØ)
GLOBAL  ARRAY          AA (1:1Ø)
EXTERNAL    ARRAY          AA (*), BB(*);
BYTE ARRAY BTYES (Ø:22) = PB ← "PB RELATIVE BYTE STRING";
LONG ARRAY   L(-5:5)
OWN ARRAY        AX(Ø:1ØØ), BX(-5:1Ø);
INTEGER ARRAY       IAX(Ø:49)=Q;
BYTE ARRAY   BAX(@)=DB+3;
```

The following examples are given to show the storage allocation code emitted for arrays declared within the scope of a procedure. Storage allocation for global arrays requires no code generation since space is allocated in the area between the last assigned DB+ location and the initial Q setting. In the following example, assume that S is positioned to point at the next available location in the stack.

1. INTEGER ARRAY IAØ (LB:UB);

```
LRA       S
SUBM      LB
DUP, INCB
STOR      Q + N      Location assigned IAØ for data label
ADDM      UB
SETR      S
```

2. INTEGER ARRAY IA1(Ø:UB);

```
LRA       S
STOR      Q + N      Location assigned IA1 for data label
LOAD      UB
INCA
ADDS      Ø
```

3. INTEGER ARRAY IA2(-5:Ø);

```
LRA       S
ADDI      5
STOR      Q + N      Location assigned IA2 for data label
ADDS      6
```

4. BYTE ARRAY BAØ (LB:UB);

```
LRA     S
LSL     1
SUBM    LB
TRBC    15
STOR    Q + N     Location assigned BAØ for byte data label
LOAD    UB
TSBC    15
SUBM    LB
ADDI    2
LSR     1
ADDS    Ø
```

5. REAL ARRAY RAØ (Ø:1ØØ);

```
LRA     S
STOR    Q + N     Location assigned RAØ for data label
ADDS    2Ø2
```

6. LONG ARRAY LAØ (-5:1Ø);

```
LRA     S
ADDI    15
STOR    Q + N     Location assigned LAØ for data label
ADDS    48
```

7. REAL ARRAY RA1 (LB:UB);

```
LRA     S
LOAD    LB
ASL     1
SUB
STOR    Q + N     Location assigned RA1 for data label
LOAD    UB
SUBM    LB
INCA
LSL     1
ADDS    Ø
```

8. INTEGER ARRAY IA3(5:15);

```
LRA   S-5
STOR  Q+N         Location assigned IA3 for data label
ADDS  11
```

DEFINE DECLARATION AND INVOCATION

Syntax --

<define declaration>	::=	DEFINE <definition list>
<definition>	::=	<define identifier> = <text> #
<define identifier>	::=	<identifier>
<text>	::=	{any sequence of valid symbols not including #}
<define invocation>	::=	<define identifier>

Examples --

Define declaration:

```
DEFINE FORI = FOR I←1 STEP 1 UNTIL #, ADDUP = A*B+C/D#
```

Text:

```
PROCEDUREX
ANYONE
IF BE THEN GO QUIT ELSE TFX(JUNK)
```

Invocation:

```
FORI
ADDUP
```

NOTE: A string containing the character # will not terminate the text for a define; i.e., the string "ABC#"# is legal text and will be terminated upon recognition of the first # not contained in a string.

Semantics --

The DEFINE declaration assigns the meaning of the defined identifiers from the ASPL source language texts.

An invocation causes the replacement of the defined identifier being invoked by the text which is associated with the defined identifier.

At declaration time, a definition is of no consequence, it has meaning only in relation to the context in which its related defined identifier appears. For this reason, undeclared identifiers may appear in definitions; all identifiers must have been declared, however, when the defined identifier is used. During compilation, syntax errors (if any) in a definition are noted following the use of the defined identifier.

Definitions can be nested; that is, defined identifiers may be used in definitions.

It is legal to use (invoke) defines in declarations, as in the example below:

```
DEFINE I = ARRAY B(Ø:1)#;
INTEGER I;  {Integer Array B(Ø:1)}
```

ENTRY DECLARATION

Syntax--

<entry declaration> ::= ENTRY <identifier list>

Examples--

```
ENTRY    SINF, COSF, TANF;
```

Semantics--

An entry declaration is provided for the specification of multiple procedure or main program entry points. If the entry declaration is local to a procedure body, the type of the procedure, if present, and formal parameter list of the procedure in which it is declared will be assigned to the entry point. An entry point must occur in the procedure body or main body at some point as a statement label, i.e. the declared identifier followed by a colon. When the statement label is encountered, the current code position is marked as the entry point for the procedure or main. All identifiers declared in an entry declaration may be used as the object of a GOTO statement, subject of course, to the same rules governing label identifiers.

Another example--

```
REAL PROCEDURE F(X); VALUE X; REAL X;
  BEGIN
    REAL     Y←1.354, Z←1.Ø E-5;
    ENTRY    F1,F2;

       F←     Y*X+Z;         <<entry point for F>>
       RETURN;
  F1:  TOS←Y*X;              <<entry point for F1>>
        GOTO L1;

  F2:  IF X<Ø.Ø THEN         <<entry point for F2>>
        TOS←REAL(TOS)-Z ELSE TOS←Ø.;

  L1:  F2←TOS;
    END <<F,F1,F2>>;
```

EQUATE DECLARATION

Syntax--

<equate declaration> ::= EQUATE <equate list>

<equate> ::= <equate identifier>=<equate expression>

<equate identifier> ::= <identifier>

<equate expression> ::= <sign><equate ref>|<equate ref>|
<equate expression><operator><equate ref>|

<equate ref> ::= <unsigned integer>|<equate identifier>

<sign> ::= +|-

<operator> ::= <addop>|*|/

Examples--

```
EQUATE    X = 1, Y = X+1, Z = Y+1
EQUATE    T = Z/2-3+X
```

Semantics--

The equate declaration assigns the value of the equate expression to the equate identifier. In the preceding examples X, Y, Z, and T would take on the values 1,2,3,-1, respectively.

INTRINSIC DECLARATION

Syntax --

<intrinsic declaration> ::= INTRINSIC <intrinsic id list>

<intrinsic id> ::= <identifier>

Semantics --

The intrinsic declaration provides a simple method for specifying system intrinsics that will be called.

Consult AMOS manual for specification of number and type of parameters, and a general discussion of the available system intrinsics.

Example --

```
INTRINSIC         FOPEN, FREAD, FWRITE;
```

LABEL DECLARATIONS

Syntax --

<label declaration> ::= LABEL <label list>

<label> ::= <identifier>

Examples --

LABEL START, STOP, NEXT

Semantics --

A label declaration defines each identifier in its label list as a label identifier.

POINTER DECLARATION

Syntax--

<pointer declaration> ::= <global pointer dec>|
<local pointer dec>

<global pointer dec> ::= <global attribute><atype>POINTER<pointer dec list>

<local pointer dec> ::= <atype> POINTER <pointer dec list>|
OWN <atype> POINTER <identifier list>|
EXTERNAL <atype> POINTER <identifier list>

<pointer dec> ::= <identifier><pointer init>|
<identifier><var reference>

<pointer init> ::= ←@<address specification>|
<empty>

<address specification> ::= <simpvar id>|
<indexed identifier reference>

Semantics--

A pointer represents the address of a quantity. That is to say, it "points" to a byte or word position. The contents of the location assigned a pointer is a word or byte data label. The type of the entity "pointed at" by the pointer is assumed to correspond to the type assigned the pointer.

An empty type attribute will assign the pointer type LOGICAL by default.

Examples--

Pointer declaration:

```
BYTE POINTER BP1,BP2=K,BP3←@A(123)
POINTER    X,Y,Z
REAL POINTER RP=IP, RP1←@X
```

PROCEDURE DECLARATION

Syntax--

```
<procedure declaration>     ::= <ctype> PROCEDURE <proc head> <proc body>

<ctype>                     ::= <type> | <empty>

<proc head>                 ::= <identifier> <formal part> <option part>|
                                <identifier>;<option part>

<formal part>               ::= (<formal param list>); <value part>
                                <specification part>

<formal param>              ::= <identifier>

<value part>                ::=  VALUE <identifier list>;|
                                <empty>

<specification part>        ::= <specification>;|
                                <specification part> <specification>;

<specification>             ::= <type> <identifier list>|
                                <atype> ARRAY <identifier list>|
                                LABEL <identifier list>|
                                <atype> POINTER <identifier list>|
                                <atype> PROCEDURE <identifier list>

<option part>               ::= OPTION <option list>;|
                                <empty>

<option>                    ::= UNCALLABLE|PRIVILEGED|
                                EXTERNAL|CHECK <integer from 0 to 3>|
                                VARIABLE|
                                FORWARD|
                                INTERRUPT   {See AMOS ERS for definition}

<proc body>                 ::= <statement>|
                                BEGIN <data group> <procedure group>
                                <compound tail>|<empty>
```

Note: If a procedure declaration appears in another procedure then it must have the option EXTERNAL.

Semantics--

A PROCEDURE declaration defines the procedure identifier as the name of a procedure and what the procedure shall be.

A procedure with a non-empty type field can be used as a function designator.

Every formal parameter must appear in the specification part.

The <value part> specifies which formal parameters are to be called by value. When a formal parameter is called by value, the formal parameter is set to the value of the corresponding actual parameter; therefore, the formal parameter is handled as a variable that is local to the procedure body. That is, any change of value of the variable will not be known outside the procedure body. Note that arrays cannot be called by value.

Only expressions may be given as actual parameters to be called by value. The expressions are evaluated once, left-to-right, in the order in which they occur in the actual parameter list.

Formal parameters that are not contained in <value part> are said to be called by reference. In this case the address of the actual parameter (which must be an identifier) is passed into the corresponding formal parameter cell. An occurrence of that parameter in the procedure may therefore change the value of the variable outside the procedure.

Procedures may be called recursively. If the procedure body is empty, the option FORWARD or EXTERNAL must be specified.

If the procedure body is non empty, it is executed when the procedure is invoked.

If the option FORWARD is specified, the complete procedure will be introduced later in the program. A FORWARD reference is required to eliminate the contradiction imposed by the requirement of procedure declaration before a procedure reference. A FORWARD reference is required when a procedure calls another procedure, which in turn references the first procedure.

If the option VARIABLES is specified, the procedure can be invoked with a variable length actual parameter list. The location (Q-4) will contain a bit mask (locations Q-5 and Q-4 if more than 16 parameters).

For each missing parameter the corresponding bit (the correspondence is from right to left) is set. Missing parameters are indicated by the actual parameter part being empty. Consecutive missing parameters starting from the right need not be represented by a sequence of commas; the occurrence of a right parenthesis in place of a comma implies the rest of the parameters are missing.

Examples --

Declaration:

```
Procedure P(A,B,C); VALUE A,B,C;
INTEGER A,B,C; OPTION VARIABLE;
```

Invocation:

P(3,,2) would generate the code:

```
LDI  3
ADDS 1
LDI  2
LDI  2
PCAL P
```

P(K) would generate the code:

```
LOAD K
ADDS 2
LDI  3
PCAL P
```

P(K1,K2,3) would generate the code:

```
LOAD K1
LOAD K2
LDI  3
ZERO
PCAL P
```

P(,,K) would generate the code:

```
ADDS 2
LOAD K
LDI  6
PCAL P
```

The following paragraphs discuss the use of pointer variables as actual parameters.

Pointers, when used as actual parameters, generate the code shown in the tables below. The advantages of having pointers as formal parameters are: indirection and indexibility.

An actual parameter may itself be a formal parameter. In this case, if as a formal parameter a pointer is a "value" pointer, the code generated is the same as that for using a globally or locally declared pointer as actual parameter.

The tables do not show the cases where the object of the pointer is a byte or multi-word. In these cases a byte load or multi-word load is generated in place of the load instruction for the object.

In addition the actual parameter may be indexed except when the corresponding formal parameter is a "reference" pointer. Where indexing is valid and the loading of an address is required, a shift instruction and an ADXA instruction <u>may</u> be generated.

<u>Actual parameter is either a declared pointer or itself a formal parameter by value</u>

<u>Actual Parameter</u>	<u>Formal Parameter</u>	
	integer	value integer
pointer	LOAD P	LOAD P,I
@pointer	LRA P	LOAD P

<u>Actual Parameter</u>	<u>Formal Parameter</u>	
	pointer	value pointer
pointer	LRA P	LOAD P

<u>Actual Parameter</u>	<u>Formal Parameter</u>	
	array	arrays may not be by value
pointer	LOAD P	

Actual parameter is itself a formal parameter by reference.

Actual Parameter	Formal Parameter: integer	Formal Parameter: value integer
pointer	LOAD P,I	LOAD P,I;LOAD S-Ø,I;DELB
@ pointer	LOAD P	LOAD P,I

Actual Parameter	Formal Parameter: pointer	Formal Parameter: value pointer
pointer	LOAD P	LOAD P,I

Actual Parameter	Formal Parameter: array	Formal Parameter: arrays may not be by value
pointer	LOAD P,I	

If the option EXTERNAL is specified, the procedure will be linked to the program at run time. EXTERNAL implies the procedure will be compiled externally in the sub-program mode.

The option UNCALLABLE can be assigned to a procedure. The definition of an UNCALLABLE procedure will be found in the ALPHA external reference specification. A procedure is assured to be callable if the option UNCALLABLE is not specified.

Examples--

```
INTEGER   PROCEDURE   LINK (A,I); VALUE I; INTEGER I; INTEGER ARRAY A;
  BEGIN
    INTEGER    L;
      WHILE    I > Ø    DO
        BEGIN
         IF    A(I+1) > KEY THEN
          BEGIN
           LINK   ←  I;
           GO QUIT;
          END;
         I  ← A(I);
        END;
     QUIT:
      END <<LINK>>;

     PROCEDURE   READ;
       IF   TAPETOG  THEN
        READFROMTAPE (NWORDS, BUF)
       ELSE
        READFROMDISK (NWORDS, SECTOR, BUF);

    PROCEDURE   T(I); VALUE I; INTEGER I;
       TOS ← LOGICAL (I) AND %377;
```

If the option PRIVILEGED is specified, then the code generated will be executed in privileged mode.

The option CHECK may be used whenever external procedures occur. The option may be used when a procedure to be used as an external procedure is declared. It may also be used when a dummy external declaration is made by the caller. The absence of the option in either case implies that no checking will take place. Otherwise, the smaller of the two parameters is used to determine the level of checking.

- Ø no checking
- 1 check procedure type only
- 2 check procedure type and number of parameters
- 3 check procedure type, number of parameters, and type of each parameter

In the case of intrinsic procedures, the level of checking is determined by the intrinsic procedure and not by the caller.

Example:

Procedure to be used as an external procedure

```
PROCEDURE P(A,B); VALUE A,B; INTEGER A,B;
   OPTION CHECK 3;
   BEGIN
      A ← A + B;
   END;
```

Procedure declared external

```
PROCEDURE P(X,Y,Z); VALUE X,Y,Z; REAL X,Y,Z; OPTION EXTERNAL, CHECK 1;
```

No error would be detected because the level of checking would be the smaller of the two parameters which is 1 and there is no conflict in procedure type.

Procedure declared external

```
PROCEDURE P(X,Y,Z); VALUE X,Y,Z; REAL X,Y,Z; OPETION EXTERNAL, CHECK 2;
```

The loader will flag an error because the number of parameters is incompatible.

SIMPLE VARIABLE DECLARATION

Syntax--

```
<simpvar declaration>          ::= <global simpvar dec>|
                                   <local simpvar dec>

<global simpvar dec>           ::= <global attribute><type><var dec list>

<global attribute>             ::= GLOBAL | <empty>

<local simpvar dec>            ::= OWN <type><nonref var dec list>|
                                   EXTERNAL <type><identifier list>|
                                   <type><var dec list>

<type>                         ::= INTEGER | LOGICAL | BYTE |
                                   DOUBLE | REAL | LONG

<var dec>                      ::= <identifier><var reference>
                                   <identifier>= X|<identifier><simpvar init>

<nonref var dec>               ::= <identifier><simpvar init>

<var reference>                ::= <empty>|
                                   =<identifier>|
                                   =<identifier><sign><USL>
                                   =<base register reference>

<base register reference>      ::= DB+<USI>|
                                   Q+<USI>|
                                   Q-<USI>|
                                   S-<USI>

<USI>                          ::= <unsigned integer >

<simpvar init >                ::= < empty>|
                                   ←<initial value >

<initial value>                ::= {a valid ASPL constant}
```

Semantics--

A simple variable declaration defines the type, addressing convention, and form of initialization of each element in the list. The type assigned a variable specifies the allowable set of ALPHA instructions which may operate on the variable.

Examples--

```
INTEGER   I,J ← 1245,L
DOUBLE    IL ← -1234579D
REAL      A,B,C ← 1.321 E -21
LONG      AA ← -1.3225 L+25, BB
BYTE      B1 ← "$", BITEV, EIGHTBITS ← 8
LOGICAL   T ← TRUE, F ← FALSE
```

SUBROUTINE DECLARATION

Syntax--

```
<subroutine declaration>   ::= <stype> SUBROUTINE <sub head><sub body>

<sub head>                 ::= <identifier><formal id part>|
                               <identifier>;

<formal id part>           ::= (<formal param list>);<value part><specification
                                                                          part>

<sub body>                 ::= <statement>

<stype>                    ::= <type> | <empty>
```

Example--

```
SUBROUTINE    RADD  (A,B,M) ; VALUE M;
  INTEGER   M;   ARRAY  A , B ;
   BEGIN
     FOR     I ←Ø STEP 1 UNTIL M DO
       A (I)←    A(I)+B(I);
     END  <<RADD>>;
```

Semantics--

A SUBROUTINE declaration defines the subroutine identifier as the name of a subroutine and defines what the subroutine shall be.

Every formal parameter must appear in the specification part.

No declarations are allowed in the scope of a subroutine body.

The value part specifies which formal parameters are to be called by value. When a formal parameter is called by value, the formal parameter is set to the value of the corresponding actual parameter; thereafter, the formal parameter is handled as a variable that is local to the subroutine body. That is, any change of value of the variable will not be known outside the subroutine body. Arrays cannot be called by value.

Only expressions may be given as actual parameters to be called by value. The expressions are evaluated once, left-to-right, in the order in which they occur in the actual parameter list.

Formal parameters not in the value part are called by reference. This means that whenever an actual parameter called by reference, appears in the procedure body, the actual parameter is re-evaluated.

Subroutines may be called recursively.

Typed subroutines can be used as a function designator.

Globally declared subroutines may be invoked only in the main program.

Locally declared subroutines may be invoked only in scope of the procedure in which they are declared.

SWITCH DECLARATIONS

Syntax --

<switch declaration> ::= SWITCH<identifier>←<label list>

Examples --

SWITCH SW ← LØ, L1, L2, L3

Semantics --

A SWITCH declaration defines an identifier to represent a set of labels. The set is composed of the identifiers in the label list.

Associated with each label in the label list (from left to right) is an ordinal number from 0 to N-1 (where N is the total number of labels in the list). This number indicates the position of the label in the label list.

The value of the switch designator corresponding to a given value of the subscript expression determines which label is selected from the label list. The identifier thus selected supplies a label in the program to which control is transferred.

V. EXPRESSIONS

EXPRESSIONS

Syntax--

<expr>	::= <T expr>|<IF expr>
<T_3expr>	::= <aexp>
<logical expr>	::= <lexp>
<IF expr>	::= <IF clause> THEN <expr> ELSE <expr>

Semantics--

Expressions in ASPL can be classified as arithmetic and logical expressions.

Execution of operators is in a left to right sequence unless there is a conflict with the hierarchy of operators in which case the hierarchy takes precedence. Operators in parentheses take precedence over operators not so contained.

For an IF expression the two expressions following THEN and ELSE must be the same size; in this context a byte expression is of length one word.

The symbols T and T_3 must be replaced as given on p.I.2.2.

The order in which operands are loaded cannot be determined in a simple manner and the user should be careful whenever side effects may be introduced by certain constructs.

e.g. A (I) ← B (I←I+1)

In this example it is not clear whether the index for the array A should be the value of I before the statement is executed or the incremented value of I. For optimization purposes the I used by the compiler is the incremented one.

A (I) ← B(I) * K + C (I←I+1)

The index for B is loaded first; for C second; for A third.

NOTE: *For long arithmetic there do not correspond any machine instructions so that all long arithmetic is done by calls on routines. For double operands multiplication and division are not allowed.*

In order to clarify the hierarchial analysis in parsing an expression, some examples are given of the order in which operators are performed. Logical and arithmetic expressions are parsed in a similar fashion but with a different set of operators.

A - B + C

operators of the same rank are performed from left to right.

A + B * C

operators of different rank are performed according to its position in the hierarchy.

(A+B) * C

operators inclosed in parentheses take precedence over operators not so inclosed.

A - B + C * D ↑ E

left to right order is maintained unless the operator is in direct contention with an operator higher in the hierarchy or is contained in parentheses.

A ↑(B-C)*D/E MOD F↑G

B-C performed first because it is inclosed in parentheses

↑ is performed next because it is of higher rank than *

* same rank as / so left to right rule applies

/ same rank as MOD

↑ higher rank than MOD

MOD last operator to be performed

ARITHMETIC EXPRESSIONS

Syntax --

<T_3 aexp> ::= <aexp> | <addop> <aexp>

<aexp> ::= <T_3 term> |
<aexp><addop><T_3 term>

<T_3 term> ::= <T_3 factor> |
<T_3 term><mulop><T_3 factor>

<T_3 factor> ::= <T_3 primary> |
<T_4 factor>↑<T_4 primary>

<T_3 primary> ::= <number> |
<T_3 variable> |
<T_3 bit expr> |
(<aexp>) | \<aexp>\ |
<T_3 function designator> |
(<T_3 assignment statement>)

<addop> ::= + | -

<mulop> ::= * | / | MOD

Semantics --

All occurrences of the symbol T_3 must be replaced consistently with one of the following words:

integer
real
long
byte
double

The symbol T_4 must be replaced with

integer
real
long

The following hierarchy holds for arithmetic operators:

1. ↑
2. *,/,MOD
3. +,-

Type mixing of operands is not allowed but type transfer functions may be used. The type of the operands will determine the type of the operation result and the type of operator used. Integer operations are used when the operands are of type byte.

The notation \<aexp>\ specifies that the absolute value is to be taken.

Examples --

```
INTEGER I,J,K;
BYTE B1,B2;
REAL X,Y;
LONG A,B,C;
DOUBLE D1,D2;
```

J+K*I	integer operators are assumed
B1+B2	integer operators are assumed
D1/D2	error: no divide instruction exists for double operands
X*Y	real operators are assumed
A-B*C	long real operators are assumed
I+B1	error: no type mixing allowed
A-X	error: no type mixing allowed

LOGICAL EXPRESSIONS

Syntax--

```
<lexp>                  ::= <disjunction>|
                            <lexp> OR <disjunction>|
                            <integer aexp><= <integer aexp><= <integer aexp>

<disjunction>           ::= <conjunction>|
                            <disjunction> XOR <conjunction>

<conjunction>           ::= <logical elem>|
                            <conjunction> AND <logical elem>

<logical elem>          ::= <logical term>|
                            <logical term><relop><logical term>|
                            <aexp><relop><aexp>
                            <byte ref><relop><byte ref><count><sdeca>|
                            <byte ref><relop>* PB <count><sdeca>|
                            <byte ref><relop><string><sdeca>|
                            <byte ref><relop>(<listelmt>)<sdeca>|
                            <T_Ø simpvar id> = <btestword>|
                            <T_Ø simpvar id> <> <btestword>

<logical term>          ::= <logical factor>|
                            <logical term><logical addop><logical factor>

<logical factor>        ::= <logical primary>|
                            <logical factor><logical mulop>
                            <logical primary>

<logical primary>       ::= <number>|<logical value>|
                            <logical variable>|
                            <logical bit expr>|
                            (<lexp>)|
                            <logical function designator>|
                            (<logical assignment statement>)|
                            NOT <logical primary>

<relop>                 ::= > | < | = | <> | >= | <=

<logical mulop>         ::= *|**|/|//|MOD|MODD
```

```
<logical addop>          ::= + | -

<byte ref>               ::= <byte pointer id><indx>|
                             <byte array id><indx>|*

<byte>                   ::= <string>|{8 bit numbers}

<count>                  ::= ,(<expr>)

<btestword>              ::= ALPHA | NUMERIC | SPECIAL

<sdeca>                  ::= <empty>|,<sdec>

<sdec>                   ::= 0|1|2

<indx>                   ::= <empty>|
                             (<expr>)|
                             (<assignment statement>)
```

Semantics--

The following hierarchy holds for logical operation.

1. NOT
2. *,**,/,//,MOD,
3. +,-
4. =,>,<>,>=,<=
5. AND
6. XOR
7. OR

The significance of the operators **, //, and MODD is the following:

```
LOGICAL  L1, L2, L3;          ZERO

    L2/L3                     LOAD   L2
                              LOAD   L3
                              LDIV,DEL
```

L1//L3	LDD L1 LOAD L3 LDIV,DEL
L2 MOD L3	ZERO LOAD L2 LOAD L3 LDIV, DELB
L1 MODD L3	LDD L1 LOAD L3 LDIV,DELB

The symbols // and MODD require that the most significant portion of the dividend has been loaded by the user if the dividend is the result of an intermediate calculation.

L2 * L3	LOAD L2 LOAD L3 LMPY,DELB
L2 ** L3	LOAD L2 LOAD L3 LMPY

The symbol ** implies that a two word result is left in the stack.

In general the result of a logical expression is left as a full word operand on top of the stack. An exception is when a relational operator is encountered in which case a value of 1 or 0 is left on top of the stack depending on whether the relation is true or false respectively. A further exception is when the result of a relational operator is used in a conditional statement in which case nothing is left in the stack and the status word is treated instead.

For the production <logical elem> ::= <byte ref><relop><byte ref><count><sdeca> it is noted that the second <byte ref> may be DB or PB relative.

The symbol $T_{\emptyset}$ must be replaced as given on p.I.2.2

Examples--

```
INTEGER K;
BYTE ARRAY BA(0:15);
BYTE POINTER BP;

IF BA(3) = BP, (5), 1 THEN     LDI     3
 K ← 1                         ADDM    BA
                               LOAD    BP
                               LDI     5
                               CMPB    3, 1  [SDEC, DB rel]
                               BNE     * + 3
                               LDI     1
                               STOR    K

IF BA <*, (6) THEN K ← Ø       LOAD    BA
                               XCH, NOP
                               LDI     6
                               CMPB    2,1 [SDEC,DB]
                               BGE     *+3
                               ZERO,NOP
                               STOR    K

IF * = BP(3),(3) THEN K ← 2    LDI     3
NOTE:  A * implies that        ADDM    BP
       an address has been     LDI     3
       loaded on the stack     CMPB    3,1[SDEC,DB]
                               BNE     * + 2
                               LDI     2
                               STOR    K

IF TOS=BP(3) THEN K ← 2        LDXI    3
NOTE:  "TOS" implies that      LDB     BP,I,X
       a value has been        CMP
       loaded on the stack     BNE     * + 3
                               LDI     2
                               STOR    K
```

IF * <> *, (-7), 2 THEN K ← 1	LDNI 7
NOTE: A negative count implies that the comparison is made from right to left	CMPB 2,1 [SDEC,DB]
	BEQ * + 2
	LDI 1
	STOR K
IF BP(2) >BP(7),(1),1 THEN K ← Ø	LDI 2
	ADDM BP
	LDI 7
	ADDM BP
	LDI 1
	CMPB 1,1 [SDEC,DB]
	BLE * + 2
	ZERO, NOP
	STOR K
IF BA(1) = "73A9", 1 THEN GO QUIT	LDI 1
ELSE GO AGAIN	ADDM BA
	LRA * + 5
	LSL 1
	LDI 4
	CMPB 1,Ø [SDEC,PB]
	BEQ AGAIN
	BR QUIT
	CON "73", "A9"
IF BA < ("73A9", %100,1), Ø	LOAD BA
THEN GO QUIT ELSE GO AGAIN	LRA *+5
	LSL 1
	LDI 8
	CMPB Ø,Ø [SDEC,PB]
	BL QUIT
	BR AGAIN
	CON "73", "A9"
	CON %40001

The following examples demonstrate some constructs which are treated as special cases:

IF TOS = ALPHA THEN GO QUIT	BTST, DEL BE QUIT
IF IVAR >= Ø THEN GO QUIT	LOAD IVAR DEL, NOP BGE QUIT
IF TOS <Ø THEN GO QUIT	TEST, DEL BL QUIT
IF DVAR = ØD THEN GO QUIT	LDD DVAR DTST,DDEL BEQ QUIT
IF TOS > ØD THEN GO QUIT	DTST, DDEL BG QUIT

An instruction which allows for a range test for integer type expressions is implemented with the following construct:

IF I1 + 2 <= I2 * I3 <= I4 THEN GO QUIT *NOTE: The syntax specifies that the only relational operator allowed is a "<=" and the only operand type is integer.*	LOAD I1 ADDI 2 LOAD I2 MYPM I3 STAX LOAD I4 CPRB QUIT
IF NOT (I1 <= I2 <= I3) THEN I1 ← 1	LOAD I1 LDX I2 LOAD I3 CPRB P+3 LDI 1 STOR I1
IF I1 <= I2 <= I3 THEN I1 ←1	LOAD I1 LDX I2 LOAD I3 CPRB P+2 BR P+3 LDI 1 STOR I1

VARIABLES

Syntax --

```
<T variable>           ::=  <T simpvar id>|
                            <T pointer id> <indx>|
                            <T array id> <indx>|
                            TOS
<integer variable>     ::=  @<T simpvar id>|
                            @<T pointer id> <indx>|
                            @<T array id><indx>|
                            ABSOLUTE (<index>)

<indx>                 ::=  <empty>|(<index>)

<index>                ::= <aexp>|<lexp>|
                           <assignment statement>
```

Semantics --

Throughout this manual whenever a reference is made to <indx> or <index> the type of the expression evaluated must be such that it can be contained in one word. i.e. integer, logical, or byte.

All occurrences of the symbol T must be replaced consistently with one of the following words:

integer
logical
byte
real
double
long

For example, the syntax above specifies that the following productions are associated with <integer variable>:

```
<integer variable>  ::=  <integer simpvar id>|@<label id>|@<proc id>|
                         <integer pointer id><indx>|
                         <integer array id><indx>|
                         @<integer pointer id><indx>|
                         @<logical pointer id><indx>|
                         @<byte pointer id><indx>|
                         @<real pointer id><indx>|
                         @<double pointer id><indx>|
                         @<long pointer id><indx>|
                         @<integer array id><indx>|
                         @<logical array id><indx>|
                         @<byte array id><indx>|
                         @<real array id><indx>|
                         @<double array id><indx>|
                         @<long array id><indx>|
                         @<integer simpvar id>|
                         @<logical simpvar id>|
                         @<byte simpvar id>|
                         @<real simpvar id>|
                         @<double simpvar id>|
                         @<long simpvar id>|
                         ABSOLUTE (<aexp>)|
                         ABSOLUTE (<lexp>)|
                         ABSOLUTE (<assignment statement>)|
                         TOS
```

The symbol @ specifies that the content of a pointer cell or the address of an array element is to be referenced.

When the symbol TOS appears in an expression, its type is determined by the context. Care must be exercised in the use of TOS in as much as the compiler will not keep a record of the number of elements previously pushed onto the stack prior to encountering the symbol TOS.

TOS is a reserved symbol which may be used on the left side of an assignment statement or in an arithmetic expression or a logical expression.

Used on the left side of an assignment statement no store instruction is generated and the result of the right side is left as the top of the stack.

If TOS occurs in an expression the contents of the top of the stack are used as an operand in the expression. The number of words used from the top of the stack depends on the type of the expression (3 for long, 1 for integer, 2 for real etc.). These are not considered to be memory locations, i.e. there is no LOAD or DUP generated before the use of the operand in contrast to variables which have been declared to have address S-Ø.

The code for the following two statements is probably not what a user may expect.

eg

```
INTEGER  I, J, R, X:
    TOS ← X ;                   LOAD   X
    K ← I + J + (R*TOS);        LOAD   I
                                ADDM   J
                                MULM   R
                                ADD,NOP
                                STOR   K
```

The use of ABSOLUTE generates the privileged instructions PLDA or PSTA depending upon whether the use appears in an expression or the left side of a replacement operator respectively.

eg

```
LOGICAL     L1, L2, L3;      INTEGER I1, I2, I3 = X;
 L1 ← ABSOLUTE (I1 * I2)                LOAD   I1
                                        MULM   I2
                                        STAX,NOP
                                        PLDA
                                        STOR   L1
```

ABSOLUTE (L2) ← I1 + 5	LOAD I1 ADDI 5 LDX L2 PSTA
ABSOLUTE (I3) ← I1 + 5 Note: I3 is the X register	LOAD I1 ADDI 5 PSTA
L1 ← ABSOLUTE (ABSOLUTE (3))	LDXI 3 PLDA STAX,NOP PLDA STOR L1
L1 ← ABSOLUTE (I3)	PLDA STOR L1

FUNCTION DESIGNATOR

Syntax--

```
<T function designator>   ::= <T procedure id><actual param part>|
                              <T subroutine id><actual param part>

<actual param part>       ::= <empty>|
                              (<actual param list>)|
                              (<stacked param list>)|
                              (<stacked param list>,
                                 <actual param list>)

<actual param>            ::= <reference param>|
                              <value param>

<reference param>         ::= <T simpvar id>|
                              <T array id> <indx>

<indx>                    ::= <empty> | (<expr>) | (<assignment statement>)

<value param>             ::= <aexp> | <lexp> | <assignment statement>

<stacked param>           ::= *
```

Semantics--

The symbol T is to be replaced with

integer
logical
byte
real
double
long

A function designator causes a previously defined function , i.e. type-procedure or subroutine, to be executed.

The function identifier references the function body which is to be executed. The <actual param part> contains a list of the actual parameters to be supplied to the function. A one-for-one correspondence must exist between the actual parameter part and the formal parameters that appear in the formal parameter part of the PROCEDURE or SUBROUTINE declaration. This correspondence is one of position, where the position of the actual parameter in the function designator corresponds to the position of a formal parameter in the declaration.

A general description of the operation of the function designator is as follows:

A. If the actual parameter is not a <stacked param>, proceed to step B. If the actual param is a <stacked param>, it is assumed that the parameter has been placed on the stack by the user prior to invoking the function designator. The user must also have allocated on the stack, immediately preceding the first <stacked param>, space for storing the value to be returned by the function.

B. The formal parameters which are named in the VALUE part of the function declaration are assigned the values of the corresponding actual parameters. These parameters are then treated as local to the function body.

C. The formal parameters not named in the VALUE part (call by reference) are replaced, whenever they appear in the function body, by the corresponding actual parameters.

D. The function body, when modified as stated above, is then entered.

NOTE: If no <stacked parameters> are specified, storage will be allocated by the compiler for the function value.

If the function designator is a formal parameter, the actual parameter part will be treated as if the formal parameter list for the function were all by reference. The number of actual parameters passed a formal function will not be checked.

BIT OPERATIONS

SYNTAX

```
<T bit expr>              ::=  <T0 bit extraction>|
                               <T0 bit concatenation>|
                               <T bit shift>

<T0 bit extraction>       ::=  <T0 primary>.(<bit extract field>)

<bit extract field>       ::=  <left extract bit>:
                               <extract field length>

<left extract bit>        ::=  <unsigned integer>

<extract field length>    ::=  unsigned integer from 1 thru 15

<T0 bit concatenation>    ::=  <T0 primary>CAT
                               <T0 primary>(<bit cat field>)

<bit cat field>           ::=  <left deposit bit>:
                               <bit extract field>

<left deposit bit>        ::=  <unsigned integer>

<T bit shift>             ::=  <T primary>&
                               <shift op>(<shift count>)

<shift op>                ::=  LSL|LSR|ASL|ASR|CSL|CSR|DASL|DASR|DLSL|DLSR|
                                            DCSL|DCSR|TASL|TASR|TNSL

<shift count>             ::=  <aexp>
```

Semantics--

The symbol T must be replaced with

integer
logical
byte
real
double
long

and $T_{\emptyset}$ must be replaced with

integer
logical
byte

In a bit extraction a contiguous bit field is extracted from the memory location of an operand. The result of the operation is a right-justified word of type integer with the leading bit set to zero. The maximum field that can be extracted in a single operation is 15 bits.

Bit concatenation is used to construct a value from fields of the specified primaries. A bit field is first extracted from the rightmost primary and deposited at the specified position in the operand to the left of the CAT operator. The result is represented as a temporary quantity and the original operands are not altered. Bit concatenation is performed from left to right.

NOTE: The shift operators are context independent in the sense that types and sizes are ignored by the compiler so if a user performs a triple shift on a single word operand a triple shift operator is emitted.

Examples--

```
INTEGER   I, J, K, M, N;  DOUBLE  D;

  I. (5:3)                      LOAD    I
                                EXF     5:3

  (I + J) . (2:4)               LOAD    I
                                ADDM    J
                                EXF     2:4

  K . (8:9)                     LOAD    K     wrap around
                                EXF     8:9   will occur

  I CAT J (3:4:5)               LOAD    I
                                LOAD    J
                                EXF     4:5
                                DPF     3:5

  D & LSR (5)                   LDD     D
                                LSR     5

  I & ASL (N+M)                 LOAD    I
                                LOAD    N
                                ADDM    M
                                STAX,NOP
                                ASL     0,X

  I & DASR (17)                 LOAD    I
                                DASR    17
```

VI. STATEMENTS

STATEMENTS

Syntax--

```
<statement>          ::= <label id>:  <statement>|
                         <compound statement>|
                         <assignment statement>|
                         <IF statement>|
                         <GO statement>|
                         <FOR statement>|
                         <CASE statement>|
                         <WHILE statement>|
                         <DO statement>|
                         <MOVE statement>|
                         <SCAN statement>|
                         <ASSEMBLE statement>|
                         <PROCEDURE call statement>|
                         <EXTN call statement>|
                         <SUBROUTINE call statement>|
                         <RETURN statement>|
                         <PUSH and SET statement>|
                         <DELETE statement>|
                         <empty>
```

ASSEMBLE - STATEMENT

Syntax--

```
<assemble statement>      ::= ASSEMBLE (<instruction slist>)

<instruction slist>       ::= <instruction>|<instruction slist>;<instruction>

<instruction>             ::= <label id> : <opcode format>|
                              <opcode format>

<opcode format>           ::= <format-1>|<format-2>|<format-3>|
                              <format-4>|<format-5>|<format-6>|
                              <format-7>|<format-8>|<format-9>|

<format-1>                ::= <memory ref opcode><address part><I-field>
                              <X-field>|<sub memref op><label id>

<sub memref op>           ::= {memory ref opcode except STOR, INCM, DECM, LDB,
                                                               LDD, STB, STD}
<address part>            ::= <var id>|<addr mode><USI255>

<var id>                  ::= <T simpvar id>|<T pointer id>|<T array id>

<addr mode>               ::= DB+|Q+|Q-|P+|P-|S-

<I-field>                 ::= ,I|<empty>

<X-field>                 ::= ,X|<empty>

<format-2>                ::= <stack opcode>|
                              <stack opcode>,<stack opcode>

<format-3>                ::= <branch subop1><arg1><I-field>|
                              <nonbranch subop1><USI31><X-field>

<branch subop1>           ::= IABZ|IXBZ|DXBZ|BCY|BNCY|CPRB|DABZ|BOV|
                              BNOV|BRO|BRE

<nonbranch subop1>        ::= {any subop1 other than <branch subop1>}

<arg1>                    ::= <label id>|P <sign><USI31>|
                              *<sign><USI31>

<format-4>                ::= <sub subop2><USI255>|
                              EXF <USI>:<USI>|DPF <USI>:<USI>
```

```
<sub subop2>      ::= {subopcode2 instr except move-ops,mini-ops,EXF,DPF}

<format-5>        ::= <mini op>

<format-6>        ::= <special op><K field>

<K field>         ::= {unsigned integer  <16}

<format-7>        ::= <subop3><USI255>|
                        PCAL <procedure id>|SCAL <subroutine id>

<subop3>          ::= {subopcode3 instr except special ops}

<format-8>        ::= <sub move op><sadmode>|
                      <sub move op><sadmode><sdec>|
                      MVBW <ccf><sdeca>|
                      <scan op>|<scan op><sdec>

<sub move op>     ::= MOVE|MVB|CMPB

<sadmode>         ::= <empty>|PB          {PB rel source address}

<sdeca>           ::= <empty>|,<sdec>

<sdec>            ::= 0|1|2

<ccf>             ::= A|N|S|AN|AS|NS|ANS

<scan op>         ::= SCW|SCU

<format-9>        ::= CON <const list>|
                      PBAR <entry id list>

<USI31>           ::= {unsigned integer <32}

<USI255>          ::= {unsigned integer <256}

<USI>             ::= <unsigned integer>

<const>           ::= <constant>|<label id>
```

Semantics--

The ASSEMBLE statement allows the user to generate code of his choice. Reference to the ALPHA instruction set will be required by the user.

The compiler will not modify P relative displacements inside an ASSEMBLE statement. Consequently, an error condition will result if a P relative address field is out of range. It will be the responsibility of the user to specify an indirect address mode if, for example, an out of range <label id> is referenced. <format-9> must be used to generate an indirect address cell. CON <label id> will enter a P relative displacement, while PBAR <entry id> will enter the PB relative address of a multiple entry point.

The ALPHA instruction set mnemonics are to be used as opcodes except for BCC (branch on condition code). Mnemonics specifying the condition code test should be used instead, as defined below:

BL -- Branch if CC=CCL ; CCF=1

BE -- Branch if CC=CCE ; CCF=2

BLE -- Branch if CC=CCL or CCE ; CCF=3

BG -- Branch if CC=CCG ; CCF=4

BNE -- Branch if CC=CCL or CCG ; CCF=5

BGE -- Branch if CC=CCG or CCE ; CCF=6

There are certain special subopcode instructions (e.g. SCAN, TNSL) which do not require an argument. These exceptions are not specified in the syntax and the compiler will accept both the omission of the argument as well as the general form specified in the syntax. (The argument will be ignored in these cases).

EXAMPLE

```
    PROCEDURE     OCTOUT (BUF,T,N); VALUE T,N;
        BYTE ARRAY BUF;
        LOGICAL  T;
        INTEGER  N;
     BEGIN <<CONVERT N DIGITS OF T TO BUF IN OCTAL>>
        LABEL LOOP;
     ASSEMBLE (
         LDX     N;
         DECX,   NOP;
         LOAD    T;
  LOOP:  DUP;
         ANDI    7;
         ADDI    %6Ø;
         STB     BUF,I,X;
         LSR     3;
         DECX;
         BGE     LOOP);
```

Another way to write it is:

```
    PROCEDURE     OCTOUT     (BUF,T,N); VALUE T,N;
       BYTE  ARRAY  BUF;
       LOGICAL  T;   INTEGER   N;
       BEGIN
         WHILE   (N←N-1)   >= Ø DO
          BEGIN
            BUF(N) ←   (T AND 7) + %6Ø;
            T ← T & LSR(3);
         END;
     END <<OCTOUT>>;
```

Example: <assemble statement>

```
ASSEMBLE (NOP, LDXA; LOAD DB+5,X; EXIT 4; CONT: ZERO, STAX)
```

Example: <format-1>

```
STB S-1,I,X  ; DECM I ;
```

Example: <format-2>

```
DDUP, DELB;
STAX
```

Example: <format-3>

```
LSL 1; BRE QUIT ;
```

Example: <format-4>

```
LDI 255; ADDI 5; EXF 7:9;
```

Example: <format-5>

```
RSW; PLDA; LLSH; PSTA;
```

Example: <format-6>

```
XEQ 4;
```

Example: <format-7>

```
PCAL READ; SCAL LOOPER; ORI @377;
```

Example: <format -8>

```
SCW 1; MVBW 1, AN; CMPB PB, 1;
```

Example; <format -9>

```
CON "**********" ;
```

ASSIGNMENT STATEMENT

Syntax--

<assignment statement>	::= <left part> ← <right part> | <left part> ← <assigment statement>
<left part>	::= <T variable> | <T variable>.(<deposit field>)
<deposit field>	::= <left deposit bit>: <deposit field length>
<left deposit bit>	::= <unsigned integer>
<deposit field length>	::= {unsigned integer from 1 through 15}
<right part>	::= <expr>

Semantics--

The assignment statement generates code for storing the value of <the right> part into the location specified by the <left part>. Type mixing between <left part> and <right part> is allowed if the number of words is the same. Type transfer functions must be used for type mixing otherwise. Note that type transfer functions must always be used for type mixing within an expression.

A deposit field specifies a subfield in the bit field of a variable. The value of the <right part> is stored into this subfield.

Examples--

```
ASSUME  INTEGER I1,I2; LOGICAL L1,L2;
        BYTE  B1,B2;   REAL R1,R2;
        LONG  LNG1,LNG2;  DOUBLE D1,D2;
        INTEGER   J = S-1, K = S-Ø;

   I1 ← I1 + I2;             LOAD  I1
                             ADDM  I2
                             STOR  I1

   TOS ← L1 * L2;            LOAD  L1
                             LOAD  L2
                             LMPY

   R1 ← R2 + FLOAT (I1)      LDD   R2
                             LOAD  I1
                             FLT,FADD
                             STD   R1

   R1 ← R2 + I1              error:  no type mixing in
                                     expressions

   R1 ← I1 + I2              error:  the number of words
                                     on the left and right
                                     sides of an assignment
                                     operator must be the same;
                                     in this context, type byte
                                     is considered as one word

   TOS ← TOS + 1             INCA

   TOS ← K + 1               DUP,INCA

   J ← J + 1                 INCB

   LNG1 ← LNG2 * LNG1        special case:  routines will handle
                                            all long operations.

   L1 ← L1 * 3               LOAD   L1
                             LDI    3
                             LMPY, DELB
                             STOR   L1

   L1 ← L1 ** 3              LOAD   L1
                             LDI    3
                             LMPY, NOP
                             STOR   L1
```

CASE - STATEMENT

Syntax--

```
<CASE statement>          ::= CASE <aexp> OF <case body>|
                              CASE *<aexp> OF <case body>

<case body>               ::= <compound statement>
```

Semantics--

The first statement in the case body has index zero.

If the index of a CASE - statement falls outside the list of statements in the case body, the <empty> statement is executed. The presence of an asterisk in a case statement means that no test will be made for an out of bounds condition

Example--

Assume INTEGER I,N;

SOURCE CODE:

```
    CASE  I  OF
BEGIN
    N ← 3;
    ;
    N ← 5;
    N ← 2;
END
```

INSTRUCTIONS:

```
LOAD I
LDI   4
STBX,LCMP
BL    *+11
BR    *+15
LDI   3
STOR N
BR    *+12
LDI   5
STOR N
BR    *+9
LDI   2
STOR N
BR    *+6
BR    *+1,X
BR    *-1Ø
BR    *+3
BR    *-9
BR    *-7
```

DELETE STATEMENT

Syntax--

```
<delete statement>          ::= DEL|
                                DELB|
                                DDEL
```

Semantics--

The delete statement generates the operators DEL, DELB, and DDEL corresponding to the above syntax. The correspondence is one-to-one.

DO STATEMENT

Syntax--

<DO statement> ::= DO <statement> UNTIL <cond clause>

Semantics--

The boolean value of <cond clause> is determined after each execution of <statement>. The statement is executed as long as <cond clause> is false. <cond clause> is defined under IF statement.

Example--

Assume INTEGER I; ARRAY A (Ø:19);

SOURCE CODE:

```
DO BEGIN I ← I-1; A(I)←2 END
   UNTIL I > 3;
```

INSTRUCTIONS:

```
DECM  I
LDI   2
LDX   I
STOR  A,I,X
LOAD  I
CMPI  3
BLE   *-6
```

EXTERNAL CALL STATEMENT

Syntax--

```
<external call statement>   ::=  EXTN (<external number><external param part>)
<external number>           ::=  <unsigned integer>
<external param part>       ::=  , <external param list>|
                                 <empty>
<external param>            ::=  <FN>|<Rsect>|<WDL>|<COUNT>
<FN>                        ::=  <string>{max of 4}|<logical id>|
                                 <logical array id>
<RSECT>                     ::=  <aexp> {logical record number in file=RSECT=Ø,1,--N}
<WDL>                       ::=  <identifier>|<identifier>(<index>)
<count>                     ::=  <unsigned integer>|<sign><unsigned integer>
```

Sematics--

The external call statement is used to invoke special system routines. The external number specifies which routine is to be called. These routines are currently defined to perform a function specified by the ALPHA breadboard and simulators. The following list defines all currently available system routines:

FUNCTION	CALLING SEQUENCE	WORDS TRANSMITTED
READ CARD	EXTN (1, <WDL>)	4Ø
PRINT LINE	EXTN (2, <WDL>)	66
READ TTY	EXTN (3, <Count>,<WDL>)	36*

FUNCTION	CALLING SEQUENCE	WORDS TRANSMITTED
WRITE TTY	EXTN (4, <WDL>)	36**
READ DISC (SERIAL)	EXTN (5, <FN>,<WDL>)	128
WRITE DISC (SERIAL)	EXTN (6, <FN>,<WDL>)	128
CLOSE DISC	EXTN (7, <FN>)	---
END SIMULATION	EXTN (8)	---
READ DISC (RANDOM)	EXTN (9, <RSECT>,<FN>,<WDL>)	128
WRITE DISC (RANDOM)	EXTN (1Ø, <RSECT>,<FN>,<WDL>)	128
WRITE TTY (1 character) From right byte of word	EXTN (11, <WDL>)	1/2
READ TAPE	EXTN (12, <WDL>)	128
WRITE TAPE	EXTN (13, <WDL>)	128
REWIND TAPE	EXTN (14)	---

** Transmit until carriage return is found in buffer or 36 words.
Rewind will write EOF on tape if previous mode was write.

If <FN>= "TAPE" for a read, write or close disk serial, an identical action will be taken as in read, write, and rewind tape.

* If <count> = Ø, read TTY for up to 36 words or carriage return.

NOTE: *WDL* *implies word data label*
FN *implies file name*
RSECT *implies random disk segment number*

FOR - STATEMENT

Syntax--

<FOR statement>	::= <FOR clause><statement>
<FOR clause>	::= FOR <integer id> ← <for element> DO| FOR * <integer id> ← <for element> DO
<for element>	::= <aexp><STEP clause><UNTIL clause>| <aexp>
<STEP clause>	::= <empty>| STEP <aexp>
<UNTIL clause>	::= UNTIL <aexp>

Semantics--

An asterisk in the <FOR clause> specifies that the subsequent <statement> is to be executed once before incrementing and testing the loop variable against the limit.

The <STEP clause> specifies the increment (or decrement) to be added to the loop variable. An empty <STEP clause> implicitly specifies an increment of 1. The value of the increment and limit will be determined only initially.

The FOR statement will be executed until the limit is exceeded.

Examples--

Assume INTEGER I, XREG = X, LIM

SOURCE CODE:	INSTRUCTIONS:
FOR I ← 3 UNTIL LIM DO A (I) ← I * 2;	LDI 3 STOR I LRA I LDI 1 LOAD LIM TBA *+2 BR *+6 LOAD I MPYI 2 LDX I STOR A,I,X MTBA *-4
FOR I ← 3 STEP 2 UNTIL 8 DO A (I) ← 2;	LDI 3 STOR I LRA I LDI 2 LDI 8 TBA *+2 BR *+5 LDI 2 LDX I STOR A,I,X MTBA *-3
FOR XREG ← I STEP -2 UNTIL LIM DO A (XREG) ← I;	LDX I LDNI 2 LOAD LIM TBX *+2 BR *+4 LOAD I STOR A,I,X MTBX *-2
FOR XREG ← 1 STEP -I UNTIL LIM DO A (XREG) ← A(I); *(Note that this is bad code because A(I) will destroy the value of the loop variable).*	LDXI 1 LOAD I NEG, NOP LOAD LIM TBX *+2 BR *+5 LDX I LOAD A,I,X STOR A,I,X MTBX *-3

GO - STATEMENT

Syntax--

```
<GO statement>        ::=  GO <label ref>|
                           GOTO <label ref>|
                           GO TO <label ref>

<label ref>           ::=  <label id>|
                           <switch id> (<index>)|
                           * <switch id>(<index>)

<index>               ::=  <aexp> | <lexp> | <assignment statement>
```

Semantics--

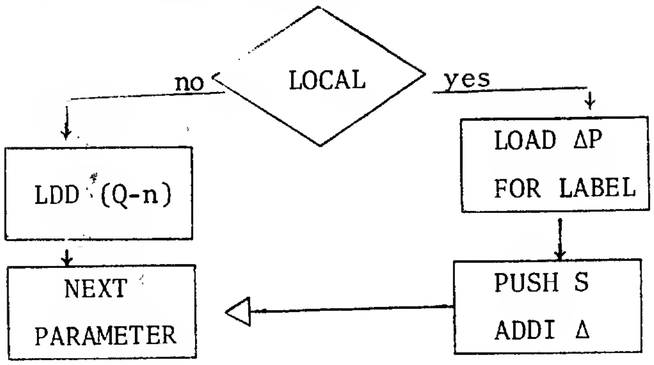
The GO - statement will execute an unconditional branch to the location specified by the label. If the label reference is a switch id, a branch to the location specified by the label in the indexed position will be executed. The first label has index zero. If the index falls outside the range of the switch declaration, control will be transferred to the next sequential statement following the GO statement. The occurrence of an asterisk in a GO statement specifies no bounds checking will be made.

Branches to labels outside of a procedure may be accomplished only by passing a label as a parameter to that procedure. A label which is a formal parameter may in turn be used as an actual parameter to another procedure.

Branches from subroutines may be made but it is the users responsibility for the integrity of the stack as the compiler takes no action to modify the stack. Of course the branches can only be made into the procedure in which the subroutine is declared or the outer block if the subroutine is global.

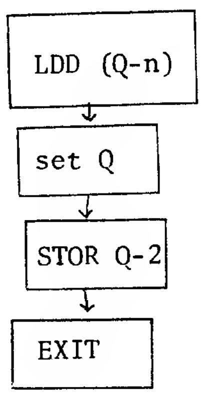
CALL TO PROCEDURE

LABEL AS ACTUAL PARAMETER



(Δ corresponds to the integer such that when added to S, the sum is equivalent to Q upon entry to the procedure)

BRANCH FROM PROCEDURE



Example --

Source Code:	Instructions:	Comment:
BEGIN		
INTEGER I;		
SWITCH SW ← L1, L2;		
GO * SW(I);	LDX I	
	BR *+12	no bounds checking
L1: I ← I + 1;	INCM I	
GO SW (I-1);	LOAD I	
	SUBI 1	
	LDI 2	# of labels in switch
	STBX, LCMP	bounds checking
	BL *+6	
L2: IF I > 3 THEN	LOAD I	
	CMPI 3	
GO FIN	BG *+2	
ELSE GO L1;	BR *-9	
FIN:	EXIT	
END;	BR *+1,X	
	BR *-12	Branch to L1
	BR *-7	Branch to L2

IF - STATEMENT

```
<IF statement>      ::= IF <cond clause><THEN part><ELSE part>
<cond clause>       ::= <cond elem>|
                        <cond clause> BOR <cond elem>
<cond elem>         ::= <cond term>|
                        <cond elem> BAND <cond term>
<cond term>         ::= <branch word>|
                        <lexp>| (<lexp >BOR < lexp>)
<branch word>       ::= CARRY | NOCARRY |
                        OVERFLOW | NOVERFLOW |
                        IABZ | DABZ |
                        IXBZ | DXBZ |<relop>
<THEN part>         ::= THEN <statement>
<ELSE part>         ::= <empty> |
                        ELSE <statement>
```

Semantics--

IF - statements allow for the conditional execution of statements. The condition is specified in <cond clause> which is evaluated as true or false. If the condition is true, the statement following the <cond clause> is executed. Otherwise, the <ELSE part> is executed.

In nested IF - statements, correspondence between the delimiters THEN and ELSE is obtained by pairing the innermost THEN with the closest following ELSE and proceeding outward.

A machine-dependent condition may be specified in the <cond clause> by using a <branch word>. The specified condition is evaluated during execution and the <statement> following the <cond clause> is executed if the condition is true. Otherwise, a branch occurs to the <ELSE part>. The branch words are described below:

<branch lexp>	Action
CARRY	Execute <THEN part> if carry bit on
NOCARRY	" " if carry bit off
OVERFLOW	" " if overflow bit on
NOVERFLOW	" " if overflow bit off
<	" " if condition code = 1
=	" " if condition code = 2
<=	" " if condition code = 1 or 2
>	" " if condition code = Ø
<>	" " if condition code = Ø or 1
>=	" " if condition code = Ø or 2
IABZ	Increment TOS. Execute <THEN part> if TOS zero
DABZ	Decrement TOS. Execute <THEN part> if TOS zero
IXBZ	Increment X reg. Execute <THEN part> if X reg zero
DXBZ	Decrement X reg. Execute <THEN part> if X reg zero

Examples--

Assume INTEGER I; LOGICAL T; LABEL QUIT;

```
    SOURCE CODE:                     INSTRUCTIONS GENERATED:

IF  T > 3 THEN I ← 21                    LOAD   T
                                         LDI    3
                                         LCMP
                                         BLE    * + 3
                                         LDI    2
                                         STOR   I

IF  LOGICAL (I) THEN                     LOAD   I
   T ← 2 ELSE GO QUIT;                   BRE    QUIT
                                         LDI    2
                                         STOR   T

IF  T  THEN GO QUIT;                     LOAD   T
                                         BRO    QUIT

IF I = 5 THEN IF T < 2                   LOAD   I
     THEN I ← 3 ELSE GO QUIT;            CMPI   5
                                         BNE    * + 6
                                         LOAD   T
                                         LDI    2
                                         LCMP
                                         BGE    QUIT
                                         LDI    3
                                         STOR   I
```

Examples (Cont.)--

```
           SOURCE CODE:                          INSTRUCTIONS GENERATED:

  IF  T  THEN  I ← 2 ELSE I ← T;                    LOAD  T
                                                    BRE   *+4
                                                    LDI   2
                                                    STOR  I
                                                    BR    *+3
                                                    LOAD  T
                                                    STOR  I

  IF  I = 5  THEN  IF T <2                          LOAD  I
     THEN I ← 3 ELSE I ← 2                          CMPI  5
      |____________|                                BNE   *+1Ø
  ELSE I ← T;                                       LOAD  T
                                                    LDI   2
                                                    LCMP
                                                    BGE   *+4
                                                    LDI   3
                                                    STOR  I
                                                    BR    *+3
                                                    LDI   2
                                                    STOR  I
                                                    BR    *+3
                                                    LOAD  T
                                                    STOR  I

  IF TOS THEN I ← 1 ELSE GO QUIT;                   BRE QUIT
                                                    LDI   1
                                                    STOR  I

  IF NOT TOS THEN TOS ← 1 ELSE I ← 1;               BRO   *+3
                                                    LDI   1
                                                    BR    *+3
                                                    LDI   1
                                                    STOR  I

  IF  =  THEN  I ← 1 ELSE T ← 3;                    BNE   *+4
                                                    LDI   1
                                                    STOR  I
                                                    BR    *+3
                                                    LDI   3
                                                    STOR  T

  IF  IABZ  THEN GO QUIT                            IABZ  QUIT

  IF  DXBZ  THEN T ← 3 ELSE TOS ← 2;                DXBZ  *+2
                                                    BR    *+4
                                                    LDI   3
                                                    STOR  T
                                                    BR    *+2
                                                    LDI   2

  IF  3 <= I <= 10 THEN GO QUIT;                    LDI   3
                                                    LDX   I
                                                    LDI   1Ø
                                                    CPRB  QUIT
```

MOVE STATEMENT

Syntax--

<MOVE statement> ::= <MOVE stmt> <sdeca>|
<MOVE-WHILE stmt> <sdeca>

<MOVE stmt> ::= MOVE <T_2 dest ref> ← <T_2 pointarr> <count> |
MOVE <T_2 pointarr> ← * <sadmode> <count> |
MOVE <T_2 pointarr> ← <string> |
MOVE <T_2 dest ref> ← (listelmt)|
MOVE <byte ref> ← <byte pointarr> <count> |
MOVE <byte ref> ← * <sadmode> <count> |
MOVE <byte ref> ← <string>
MOVE <byte pointarr> ← <number list>

<T_2 dest ref> ::= <T_2 pointarr> | *

<T pointarr > ::= <T pointer id> <indx> |
<T array id> <indx>

<count> ::=,(<expr>)

<sadmode> ::= <empty> | PB {PB rel source address}

<byte ref> ::= <byte pointarr> | *

<MOVE-WHILE stmt> ::= MOVE <byte ref> ← <byte ref> WHILE <ccf>

<ccf > ::= A|N|S|AN|AS|NS|ANS

<sdeca> ::= <empty>|,<sdec>

<sdec> ::= Ø|1|2

Semantics--

The symbols T and T_2 must be replaced as given on p.I.2.2.

An asterisk specifies that the stack has been loaded with the appropriate address.

The <ccf> values are: N = numeric
A = alphabetic
S = upshift

Any combination of ccf values is allowable.

The entity <sdeca> specifies by how much the stack is to be decremented at the end of the move instruction. The default value is to restore the stack to its setting before the MOVE statement. Thus an empty <sdeca> implicitly specifies that the stack is decremented by 3 after a <MOVE stmt> and by 2 after a <MOVE-WHILE stmt>.

The source address in a MOVE or MVB instruction may be DB or PB relative. The destination address can only be DB relative. It is noted that only local arrays can be PB relative.

Examples--

```
Assume  INTEGER I, K;
        INTEGER ARRAY  IAY (Ø:1Ø);
        LOGICAL ARRAY  LAY (Ø:1Ø);
        BYTE    ARRAY  BAY (Ø:1Ø), BAYPB (*) = PB ← "STRING"
        INTEGER POINTER IP;
        LOGICAL POINTER LP;
        BYTE    POINTER BP;
```

SOURCE CODE:	INSTRUCTIONS:
MOVE IAY (Ø) ← IAY (1),(5);	LDXI Ø LRA IAY,I,X LDXI 1 LRA IAY,I,X LDI 5 MOVE 3
MOVE IAY (K) ← LAY (I*3),(K+1);	LDX K LRA IAY,I,X LOAD I MPYI 3 STAX,NOP LRA LAY,I,X LOAD K ADDI 1 MOVE 3.
MOVE LP ← IP,(K), 1;	LOAD LP LOAD IP LOAD K MOVE 1
MOVE BP ← * WHILE ANS;	LOAD BP XCH,NOP MVBW 2,7 [SDEC,CCF]
MOVE BAY (2) ← BAYPB (1), (4), 2;	LDXI 2 LRA BAY,I,X LRA BAYPB LSL 1 ADDI 1 LDI 4 MVB PB,2

PROCEDURE CALL STATEMENT

Syntax--

<procedure call statement> ::= <procedure id> <actual param part>

Semantics--

A procedure call statement causes a previously defined procedure to be executed.

The procedure identifier references the procedure body which is to be executed. The <actual param part> contains a list of the actual parameters to be supplied to the procedure. A one-for-one correspondence must exist between the actual parameter part and the formal parameters which appear in the formal parameter part of the procedure declaration. This correspondence is one of position, where the position of the actual parameter given in the procedure call statement corresponds to the position of a formal parameter in the PROCEDURE declaration.

A general description of the operation of the procedure call statement is as follows:

A. If the actual parameter is not a stacked param, proceed to step B. If the actual param is a stacked param, it is assumed that the parameter has been placed on the stack by the user prior to invoking the procedure call statement.

B. The formal parameters which are named in the VALUE part of the procedure declaration are assigned the values of the corresponding actual parameters. These parameters are then treated as local to the procedure body.

C. The formal parameters not named in the VALUE part {call by reference} are replaced, whenever they appear in the procedure body, by the corresponding actual parameters.

D. The procedure body, when modified as started above, is then entered.

NOTE: If the procedure called is a function designator, the function value space is deleted following the call.

If the procedure identifier is a formal parameter, the actual parameter part will be treated as if the formal parameter list for the procedure were all by reference. The number of actual parameters passed a formal procedure will not be checked.

Examples

```
SPAN (*,A(2Ø), 1.325)
CALCULATE
SINF (ARG)
```

PUSH and SET STATEMENT

Syntax--

<push registers>	::= PUSH (<register spec list>)
<set registers>	::= SET (<register spec list>)
<register spec>	::= S|Q|X|STATUS|Z|DL|DB

Semantics--

These statements specify a push or set of the registers as defined by the instructions PSHR and SETR in the ALPHA 1 EXTERNAL REFERENCE SPECIFICATIONS. The register spec list specifies which registers will be selected. The push registers statement will generate code to push the contents of the specified registers onto the stack as defined by the instruction PSHR. The set registers statement will generate code to pop values from the stack into the selected registers as defined by the instruction SETR. Note that SETR, hence set registers, requires privileged mode when registers STATUS, DB, DL, or Z are specified. The user must have loaded the stack with values prior to performing a set registers statement. The compiler assumes the user has loaded the proper values in the proper sequence.

Note: The Alpha breadboard and simulators use an outdated register specification sequence (old version of SETR and PSHR instructions in ALPHA ERS). This sequence is: <register spec> ::= S|Q|Z|STATUS|Z|MASK|X

Examples--

PUSH (S,Q,Z);	emits	PSHR %23
SET (S,Q);	emits	SETR 3

RETURN - STATEMENT

Syntax--

```
<return statement>        ::=  RETURN <count>

<count>                   ::=  <unsigned integer>|
                               <empty>
```

Semantics--

The return statement is used to generate an EXIT in the body of a procedure or an SXIT if in the body of a subroutine. The count field specifies the amount by which the stack will be decremented. An empty count field specifies use of a decrement value corresponding to the number of words required in the formal parameter part.

Example--

```
PROCEDURE   P(A,B); VALUE A; DOUBLE A; INTEGER I;
  BEGIN
    RETURN  1; ---      EXIT 1
    RETURN     ---      EXIT 3
  END
```

SCAN STATEMENT

Syntax--

<SCAN statement> ::= <SCAN-WHILE stmt><sdeca>|
<SCAN-UNTIL stmt><sdeca>

<SCAN-WHILE stmt> ::= SCAN <byte ref> WHILE <testword>

<byte ref> ::= <byte pointer id><indx>|
<byte array id><indx>
*

<testword> ::= <T_1 simpvar id>|<integer>|
"<char><char>"

<SCAN-UNTIL stmt> ::= SCAN <byte ref> UNTIL <testword>

<sdeca> ::= <empty>|,<sdec>

<sdec> ::= 0|1|2

Semantics--

The symbol T_1 is to be replaced by:

integer
logical

The symbol * indicates that the stack has been loaded with the appropriate address.

An empty <sdeca> implicitly specifies that the stack is to be decremented by 2 at the end of the scan instruction.

Examples--

Assume BYTE ARRAY BAY (Ø:1Ø);
INTEGER I;

SOURCE CODE:	INSTRUCTIONS:
SCAN * WHILE I;	LOAD I SCW 2 [SDEC]
SCAN BAY (2) UNTIL I;	LDXI 2 LRA BAY,I,X LOAD I SCU 2 [SDEC]

SUBROUTINE CALL STATEMENT

Syntax--

<subroutine call statement> ::= <subroutine id><actual param part>

Semantics--

A subroutine call statement causes a previously defined subroutine to be executed.

When a subroutine is called, the Q-register remains unchanged. Thus, a subroutine body can contain non-local references to variables declared within the scope of the procedure in which the subroutine is declared. These variables, of course, must have been declared prior to the subroutine declaration.

In other respects, the operation of the subroutine call statement is the same as the operation of the procedure call statement.

WHILE - STATEMENT

Syntax--

<WHILE statement> ::= WHILE <cond clause> DO <statement>

Semantics--

The boolean value (true or false) of the <cond clause> is determined before each execution of <statement>. The statement will be executed as long as <cond clause> is true. <cond clause> is defined under IF statement.

Examples--

Assume INTEGER I; ARRAY A (Ø:1Ø);

SOURCE CODE:

```
WHILE I > 3 DO
BEGIN
   I ← I-1;
   A(I) ← 2
END;
```

INSTRUCTIONS:

```
LOAD   I
CMPI   3
BLE    *+6
DECM   I
LDI    2
LDX    I
STOR   A,I,X
BR     *-7
```

VII. APPENDIX

TYPE TRANSFER FUNCTIONS

Syntax--

```
<type-transfer function designator>    ::= FIXT (<R>)|
                                            FIXR (<R>)|
                                            BYTE (<L>)|
                                            BYTE (<I>)|
                                            REAL (<I>)|REAL (<D>)|
                                            REAL (<LNG>)|
                                            DOUBLE (<I>)|DOUBLE (<L>)|DOUBLE (<B>)|
                                            LONG (<R>)|LONG (<D>)|
                                            INTEGER (<L>)|INTEGER (<B>)|INTEGER (<D>)|
                                            LOGICAL (<I>)|LOGICAL (<B>)

<I>      ::=  expression of type integer
<R>      ::=  expression of type real
<L>      ::=  expression of type logical
<B>      ::=  expression of type byte
<D>      ::=  expression of type double
<LNG>    ::=  expression of type long
```

Semantics--

Since type mixing in expressions will result in a syntax error, it is necessary to use type transfer functions if any mixed expressions are to be evaluated. Careful attention to variable type will be essential. Type transfer functions will be required to produce homogeneity within expressions. The available type transfer functions are the following:

BYTE (I or L)	treat integer or logical argument as type byte
FIXT (R)	fix and truncate a real argument leaving a double result
FIXR (R)	fix and round a real argument leaving a double result
REAL (I or D or LNG)	convert integer, double or long argument to type real
INTEGER (L or B or D)	treat logical or byte as integer at compile time or convert a double argument to type integer
LOGICAL (I or B or D)	treat integer or byte or double argument as logical at compile time
DOUBLE (I or B)	convert single word to double
DOUBLE (L)	no code emitted
LONG (R or D)	convert a real argument to type long or a double argument to type long

Examples--

```
ARRAYX(INTEGER(FIXT(R1+R2)))
REAL(DOUBL)
INTEGER(BYT)+2
INTEGER(L1+2)
LOGICAL(I*J)
INTEGER(A>B)
INTEGER(TRUE)
BYTE(I+J)
```

RESERVED SYMBOLS

The following list of symbols is reserved for compiler use, and may not be used to name an identifier:

ALPHA	END	LOGICAL	SINGLE
AND	ENTRY	LONG	SPECIAL
ABSOLUTE	EQUATE	MOD	STEP
ARRAY	EXTN	MODD	SUBROUTINE
BEGIN	FALSE	MOVE	SWITCH
BYTE	FIXR	NOT	THEN
CARRY	FIXT	NUMERIC	TO
CAT	FOR	OPTION	TOS
COMMENT	FORWARD	OR	TRUE
DABZ	GO	OVERFLOW	UNCALLABLE
DDEL	GOTO	POINTER	UNTIL
DEFINE	IABZ	PROCEDURE	VALUE
DEL	IF	PUSH	WHILE
DELB	INTEGER	REAL	WITH
DOUBLE	INTERRUPT	RETURN	XOR
DXBZ	IXBZ	SCAN	INTRINSIC
ELSE	LABEL	SET	BOR
BAND			

The following list of symbols are used to specify register references.

DB Q S X PB

Error # (decimal)	Meaning
Ø	Overflow: integer representation > 32 bits
1	Overflow: Double or real representation > 32 bits: Composite or based number.
2	Illegal integer representation.
3	Non-integral result.
4	Number base 2 > N > 16.
5	Missing or incorrectly places "]" in a based number representation.
6	Missing "/" in composite integer.
7	Missing "]" in composite integer.
8	Conversion error: real or long.
9	ASCII conversion field size error.
1Ø	Expects integer type.
11	Missing ":" in bounds specification.
12	Missing exponent in read or long number.
13	Erect label identifier.
14	Illegal assemble form.
15	Illegal identifier in specification part.
16	Illegal PB rel array dec. restricted to local one only.
17	Illegal result in address calculation.
18	PB relative arrays are restricted to type BYTE, LOGICAL, or INTEGER.
19	Illegal direct array specification can only be DB/Q.
2Ø	Illegal mode/level combination. DB must be used with global arrays; Q with local arrays.
21	Illegal DB allocation.
22	PB relative arrays must be statically initialized.

Error # (decimal)	Meaning
23	Illegal initialization element.
24	Bounds must be undefined "*" when reference is specified.
25	Only global or PB relative arrays may be statically initialized.
26	Illegal equate element.
27	Identifier match failure - actual parameter part.
28	Illegal address in initialization list.
29	Multiply defined forward declaration.
30	Declarations not allowed in subroutine body.
31	No parameters allowed in interrupt procedures.
32	Illegal nested procedure declaration. Restricted to external procedures only.
33	Illegal attribute.
34	Illegal formal parameter: subroutines are not defined in formal parameter definition.
35	Missing specification for formal parameter.
36	Dynamic bounds not allowed for global arrays.
37	DB relative counter >255.
38	Q+ relative counter >127.
39	Missing ")"
40	Missing "(".
41	Illegal reference parameter.
42	Expected a procedure identifier.
43	Illegal statement terminator. {BEGIN}.
44	Illegal statement terminator.
45	Missing ":"
46	Undeclared identifier.
47	Illegal opcode mnemonic.
48	Expected constant or procedure ID.
49	Illegal statement beginner.
50	Missing ":" in extract or deposit field specification.
51	Actual and formal parameter lists do not match in length.

Error # (decimal)	Meaning
52	Value size stack error.
53	Type procedure required when called from expression.
54	Expect an entry identifier.
55	Too many formal parameters: Ø≤FP≤31
56	(Ø:<simpvar id>) only allowed.
57	No dynamic lower bound allowed this version.
58	PB - relative array used as actual parameter of reference.
59	Illegal register reference: displacement exceeds maximum for specified register and mode.
6Ø	Illegal register reference: X± <unsigned integer> not valid.
61	Displacement out of range for specified register and mode.
62	Illegal specification for interrupt attribute.
63	Illegal declarator ordering or too many declarators.

ASPL1 ERROR #	
64	
65	Phase 2. Auxtab Overflow.
66	Phase 2.
67	Emit illegal mem-reference instruction
68	Illegal IF stmnt.
69	Illegal GO stmnt.
70	Illegal Switch.
71	Illegal WHOLE stmnt.
72	Illegal Variable reference in MOVE-or-SCAN stmnt.
73	Illegal MOVE stmnt.
74	Illegal MOVE (MOVE-WHOLE).
75	Illegal SCAN stmnt.
76	Phase 2.
77	Phase 2.
78	Phase 2.
79	Phase 2.
80	Phase 2. Referenced undeclared label.
81	Phase 2. (Phase check).
82	Phase 3.
83	Phase 5.
84	Phase 5.
85	Illegal MOVE stmnt.
86	
87	Illegal Conditional Clause.
88	Illegal FOR stmnt.
89	Illegal CASE stmnt.
98	Illegal external number.
99	Reserved symbols may not be redefined.
100	Multiply defined identifier.

ASPL1 ERROR #	
129	Illegal type.
130	Error in partial word designator.
131	Invalid index.
132	Error in shift designator.
133	Type incompatibility in expression.
134	Store into type procedure identifier out of scope.
135	Error in catenate operator.
140	Error in type transfer function.
141	Error in fix instruction or absolute value.
144	Error in emitting memory operator.
147	Illegal operator.
150	Type incompatibility.
152	Illegal long operator.
160	Illegal class for variable.
161	Illegal type for variable.
162	Size incompatibility in assignment.
163	Expects assignment operator.
165	Wrong type stored into X register.
166	Extract field from illegal type.
170	Illegal primary.
171	Expects left paren in expression.
172	Missing right paren in expression.
173	Illegal constant.
174	Expects right paren in type transfer.
175	Expects asterisk.
176	Expects relational or comma.
182	Missing left paren in count for byte compare.
183	Missing right paren in count for byte compare.
184	Expects constant for sdec field.
185	Constant too large in sdec field.
200	Recursive define.
201	MAX STRING LENGTH EXECUTED.
202	Illegal list separator or terminator.
205	SYMBOL TABLE OVERFLOW.
999	Built in loop in define invention terminated.

DISTRIBUTION LIST

ASPL

1. Aggarwal, Naresh
2. Andrews, Harlan
3. Ansley, Bill
4. Athearn, Fred
5. Bain, Mitch
6. Baker, Rick
7. Bale, Jon
8. Barraza, Dave
9. Basiji, Jim
10. Bausek, Gerry
11. Bellizzi, Bob
12. Bergh, Arne
13. Berte, Bill
14. Birenbaum, Larry
15. Blease, Tom
16. Bollinger, Lee
17. Bond, Bob
18. Branthwaite, Terry
19. Brawn, Mel
20. Brigante, Vince
21. Brown, Steve
22. Candlin, Jim
23. Chernack, Charlie
24. Citragno, Allen
25. Crandall, Ron
26. Dieckman, John
27. Duley, Jim
28. Eccols, Mike
29. Edwards, Sam
30. Ellestad, Tom
31. Eskedal, Ole
32. Forbes, Bert
33. Foster, Bill
34. Gabet, Jean-Michel
35. Gapp, Gary
36. Gard, Bill
37. Gibbons, Dan
38. Green, Bob
39. Green, Mike
40. Green, Barney
41. Haccou, Bill
42. Haccou, Waldy
43. Hackborn, Dick
44. Hacker, Marvin
45. Hamilton, Jim
46. Hamm, Terry
47. Helness, Karl
48. Hewer, Alan
49. Hewlett, Jim
50. Holden, Bill
51. Holl, Jim
52. Holland, Ed
53. Huffstetter, Bob
54. Hunt, Tony
55. Jacobs, Denis
56. Jacobs, Jake
57. Jeung, Doug
58. Johnson, Dave
59. Johnson, Lee
60. Jorn, Fritz
61. Katzman, Jim
62. Kieser, Earl
63. Kintz, Joe
64. Kline, Harry
65. Klingman, Ken
66. Kornei, Tom
67. Lovlien, Dick
68. Lundervold, Dag
69. Lyman, Dick
70. MacDonald, Jack
71. Mallory, Rosie
72. Marston, Al
73. Matsumoto, Ron
74. McEvoy, Dennis
75. McGuire, Dix
76. McIntire, Dick
77. Miller, Ed
78. Miller, Jim
79. Miller, Micki
80. Mintz, Ken
81. Mintz, Stan
82. Mitchell, Larry
83. Mixsell, Joe
84. Moley, Dick
85. Murrin, Bill
86. Neill, Malcolm
87. Ng, Steve
88. Noonan, Jack
89. Nordgren, Brett - Rockaway, N.J.
90. O'Briant, Jim
91. Opdendyk, Terry
92. Pavone, John
93. Pearson, Richard
94. Pocek, Ken
95. Radvany, Imre
96. Riesenberg, Don
97. Riggins, Cle (2)
98. Robinson, Claude
99. Rosenfeld, Paul
100. Russell, Jim
101. Sanders, Dave
102. Schroeder, Ken
103. Shanks, Willis
104. Shipman, John
105. Smith, Jerry
106. Stutes, Earl

DISTRIBUTION LIST

ASPL (Cont.)

107. Summers, Lloyd
108. Taraldson, Jim
109. Toepfer, Dick
110. Toschi, Elio
111. Unanski, Bob
112. Vallender, Steve
113. Wacken, Cliff
114. Wallace, Scott
115. Welsch, John
116. White, Fred
117. Winn, Denis
118. Wolff, John
119. Wolff, Walter
120. Zeissler, Myron
121. Magleby, Kay - 5U
122. Cook, Dick - 8U
123. Bobroff, Jim - 8U
124. Ray, Bill - 5U
125. Rytand, Bill - 5U
126. Anderson, Dick - 51U
127. Howard, Dan - Colorado Springs
128. Negrete, Marco - Loveland
129. Watson, Bob - Loveland
130. Blokker, John - Rockaway, N.J.
131. Siegel, Floyd - San Diego
132. Stoft, Paul - 1U
133. Monnier, Dick - 30
134. Levin, Mort - Waltham
135. Miller, George - Avondale
136. Ohme, Wolfgand - GmbH
137. Fong, Art - YHP
138. Roberts, Gordon - HP Ltd.
139. Chambreau, Mike - 11L